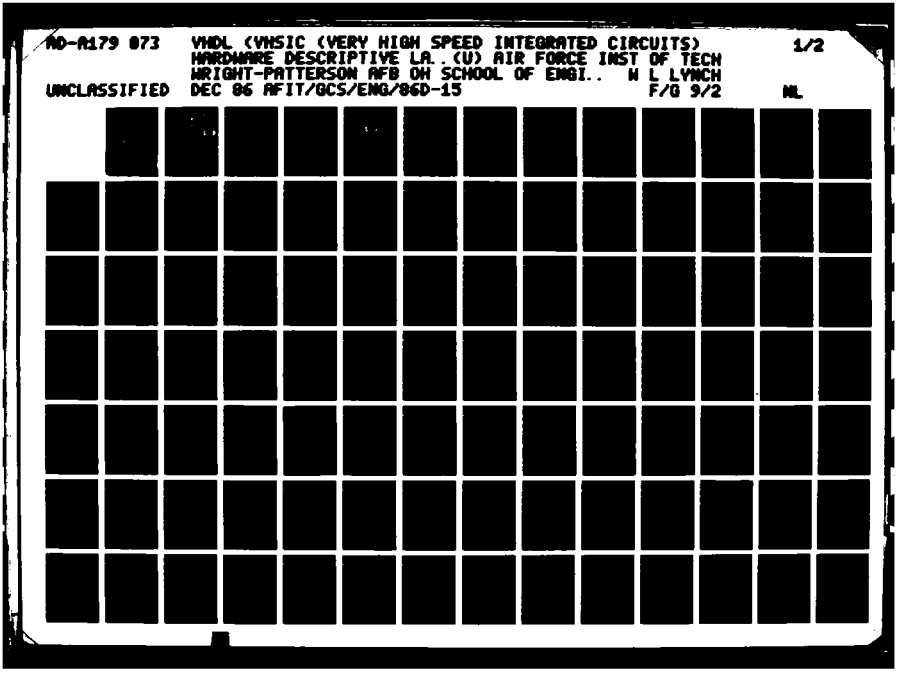
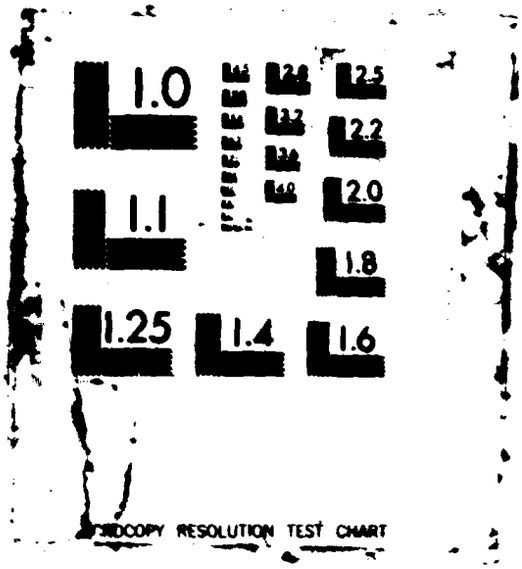

AD-A179 073
VHDL (VHSIC (VERY HIGH SPEED INTEGRATED CIRCUITS)
HARDWARE DESCRIPTIVE LA..(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. W L LYNCH
UNCLASSIFIED
DEC 86 AFIT/GCS/ENG/86D-15
F/G 9/2
1/2
NL


1.0
1.1
1.25
1.4
1.6
2.0
2.2
2.5
RESOLUTION TEST CHART

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AD-A179 073

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release; Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/86D-15	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) VHDL Prototype Simulator	
12. PERSONAL AUTHOR(S) William L. Lynch, Major, USA	
13a. TYPE OF REPORT MS Thesis 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Year, Month, Day) 86 Dec 15. PAGE COUNT 129
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	VHDL Simulator Simulation Model HARDWARE Description Language
09	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Thesis Advisor: Harold Carter

LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AIC)
Wright-Patterson AFB OH 45433

Approved for public release IAW AFR 190-1
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology
Wright-Patterson AFB OH 45433

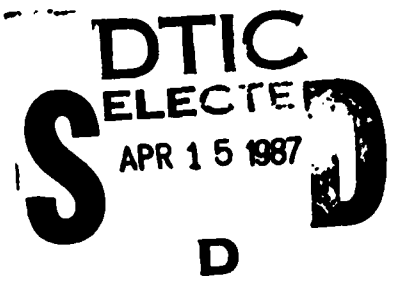
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Professor Harold Carter	22b. TELEPHONE (Include Area Code) 513-255-5533 22c. OFFICE SYMBOL AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE

Abstract

This research effort studied, modeled and implemented a prototype simulator kernel for VHDL in a UNIX environment. The prototype simulator was written in the C programming language. The detailed design of the prototype simulator includes an installation guide, users manual, and design description. The simulation program, called "VSIM" for VHDL Simulator, does not support the full capabilities of VHDL. It implements the simple signal assignment statement and models transport and inertial delay. Requirement specifications are given for both the prototype and a fully implemented VHDL simulator.

AFIT/GCS/ENG/86D-15



VHDL PROTOTYPE SIMULATOR

THESIS

William Leo Lynch Jr.

Major, USA

AFIT/GCS/ENG/86D-15

AFIT/GCS/ENG/86D-15

VHDL PROTOTYPE SIMULATOR

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

William Leo Lynch Jr.

Major, USA

December 1986

Preface

This thesis was primarily concerned with the development of a prototype simulator kernel for VHDL that operates in a UNIX environment. The program implements the major basic functions of VHDL using the C programming language.

The prototype provides proof of design concept and establishes a baseline for the continued development of a simulator in the AFIT VHDL environment. It also provided a unique opportunity to study hardware simulations, hardware description languages (VHDL) and the UNIX Operating System. It was a tremendous learning experience in both software design and engineering.

I want to express my appreciation to those many people - fellow VHDL group members, thesis committee members and special friends - who contributed so much to making this effort successful. My sincere thanks and gratitude to Professor Hal Carter, my thesis advisor, who made it all possible by suggesting this project and through the guidance, support and encouragement he provided throughout the project.

I want to especially thank my wife, Elia, and children, Bernadette and Vicente, for their unrelenting support and encouragement. Most importantly I am deeply indebted to

them for their understanding in once again enduring the hardships of a prolonged separation.

Finally, my words of praise would be incomplete without mentioning, Linda Burnett, my typist, who spent many long hours often on short notice typing this thesis and deciphering my cryptic handwriting. Thank you!

Table of Contents

Figure List

Abstract

This research effort studied, modeled and implemented a prototype simulator kernel for VHDL in a UNIX environment. The prototype simulator was written in the C programming language. The detailed design of the prototype simulator includes an installation guide, users manual, and design description. The simulation program, called "VSIM" for VHDL Simulator, does not support the full capabilities of VHDL. It implements the simple signal assignment statement and models transport and inertial delay. Requirement specifications are given for both the prototype and a fully implemented VHDL simulator.

Keywords: Hardware Description Languages, Theses, Requirements,

VHDL PROTOTYPE SIMULATOR

I. Introduction

1.1 General

The function, speed and performance that can be achieved from a single integrated circuit has increased greatly in recent years due to the rapid advancement in silicon fabrication technology. The increasing functional capabilities of integrated circuits have necessarily been accompanied by increasingly complex circuit designs which limit the individual designer's ability to fully understand the circuit design. This trend toward ever more complex designs, especially in very large scale integrated circuits, has necessitated the use of design teams in the development of very large scale integrated circuits. The design team concept has created the problem of how to communicate design information concisely, accurately, and efficiently.

One method used to describe and model electronic circuits is the formal language approach, commonly known as

Hardware Description Languages (HDL). Most HDLs in use today were developed when integrated circuit function was limited to small and medium scale circuits. Unfortunately, these languages are not adequate for the designer's needs in the development of today's state-of-the-art very large scale and very high speed integrated circuits (VLSI, VHSIC). To keep pace with the development of increasingly more complex integrated circuit designs, new software tools need to be developed that can model and simulate complex integrated circuit designs in a concise and timely manner.

Computer aided design is an important tool in the engineering design process. Sophisticated computer programs are now widely used to accomplish the menial and time consuming tasks associated with hardware developments. The design of digital logic circuits has been aided and improved by the use of modern computer simulation techniques. Hardware simulators are used today in government, industry and education to build and exercise models of a digital circuit on a computer. These programs range from simple simulation routines to highly sophisticated and complex program systems.

A hardware simulator* allows the designer to take a high level design and exercise its operations. Through the

---

* As used throughout this thesis, a hardware simulator is one that simulates, usually before construction, hardware designs.

use of a simulator, the designer can create a software model for a network, exercise the model with a set of inputs and observe the output of the network at various test points. The response of the simulator in terms of predicted signal values versus time should closely correspond with the response of the actual circuit modelled to be of practical value (1:63).

Most design projects use only one type of hardware simulation: gate-level logic simulators. Logic simulation is an especially invaluable tool for VHSIC class circuits since design errors are very costly and breadboarding is impractical (1:63).

1.2 Background

State-of-the-art developments in the electronics industry of the 1960's were driven by the needs of the military. Industry was encouraged to pursue Department of Defense (DOD) business since integrated circuits being developed for the military were also of sufficient general purpose to be applicable to commercially marketable products. In this environment, private industry benefitted from DOD's research and development expenditures. This is no longer the situation since the DOD share of the total integrated circuit market has decreased to approximately twenty percent. The military is no longer the major user of

advances in semiconductor technology. It has become one user among many, and its impact upon the semiconductor industry has decreased. Additionally, DOD has increasingly sought more complex and special-purpose integrated circuits which has moved DOD into an increasingly specialized sector of the marketplace (2).

The divergence of military and civilian applications - the military's desire for high speed signal processors versus the more general purpose processors required for commercial use - caused DOD planners to believe that industry could not be expected to develop integrated circuits for military use in a timely manner (3). As a result, the DOD established the Very High Speed Integrated Circuit (VHSIC) technology development program in 1980 to encourage the development of integrated circuit technology for military uses. DOD's hope was that once the technology became available, civilian applications would be found that would complement future military needs, thereby further stimulating research and development. The major goals of the VHSIC program are: 1) to develop the technology necessary to produce submicron devices; 2) to increase processing throughput; and 3) to formulate the new circuit design methodologies and computer aided design (CAD) tools required to make maximum use of the new technology (4:94-102).

The advantages gained in the reduction of system size, weight, and power requirements through VHSIC class integrated circuits over current technology is expected to both reduce the cost and increase the reliability and maintainability of new electronic systems. Current plans also call for the use of VHSIC technology in upgrading existing weapon systems (4:94). It is envisioned that one VHSIC chip could have 250,000 or more logic gates. The complexity of design and associated high cost require the design validation of VHSIC chips prior to manufacture.

Because the available simulation languages did not have the capabilities to adequately model VHSIC class chips, the VHSIC Program Office at Wright-Patterson AFB funded the development of a VHSIC Hardware Description Language (VHDL) to meet current and projected VHSIC application needs and to facilitate the transfer of VHSIC technology. Based on Ada, VHDL incorporates such VHSIC specific requirements as portability, maintainability, timing and the ability to hierarchically model and simulate designs.

The VHSIC Program Office is sponsoring research at the Air Force Institute of Technology (AFIT) to develop a UNIX-based VHDL system called the AFIT VHDL Environment (AVE). A simulation tool driven by VIA, the AFIT intermediate format, which is the central data base for AVE, is a part of the AFIT VHDL effort. The purposes of the simulator are to assist the VHSIC Program Office's

evaluation of VHDL for clarity of syntax, and to verify ease of use for the designer in describing VHSIC class chips.

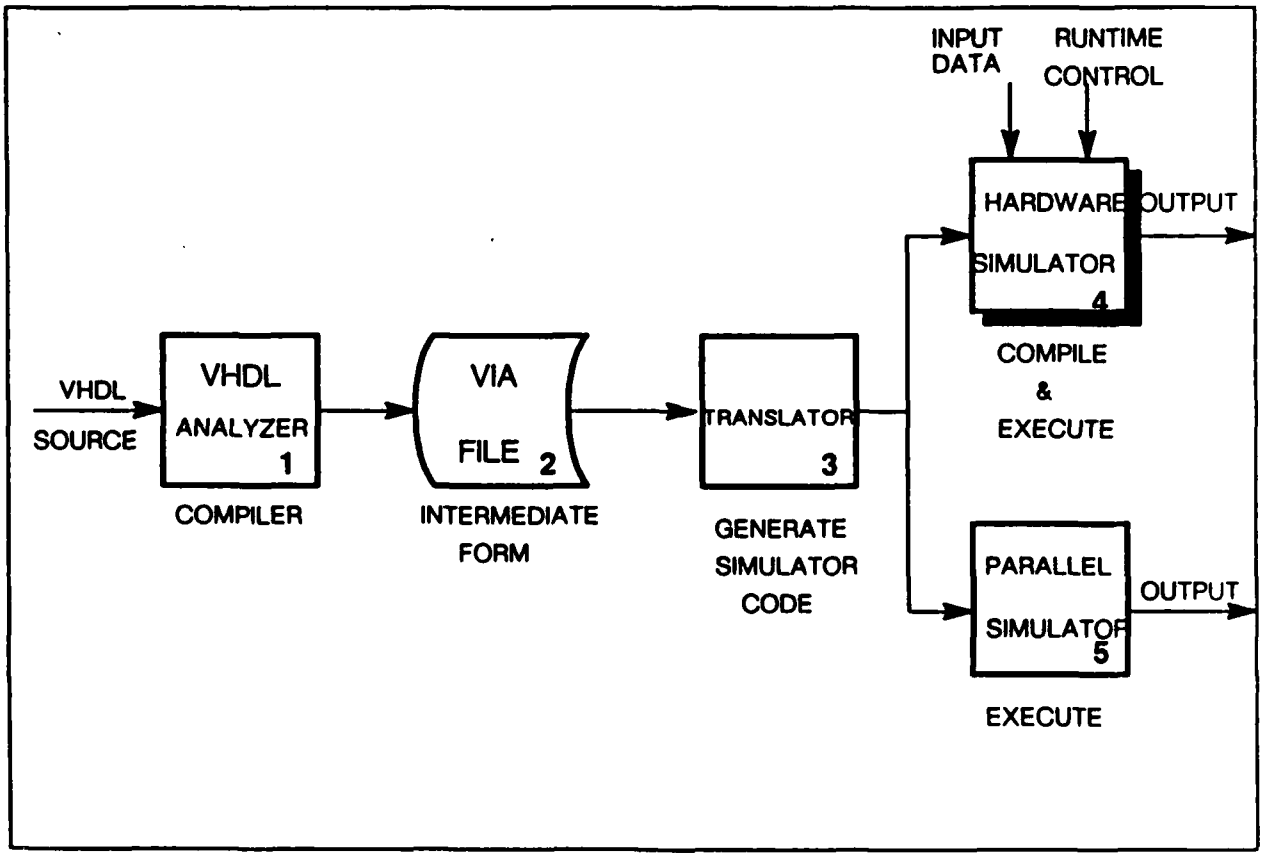


Figure 1-1 AFIT VHDL Environment

Figure 1-1 shows the relationship of this thesis, the development of a hardware simulator (highlighted), to the AFIT VHDL research effort. Research at AFIT is directed at developing a system using as input the VHDL language. The

system is being designed to assist engineers in academia in the specification, design and validation of hardware components and systems. Although nonportable to other than the UNIX operating system, the goal of the system is to operate fast and provide the full capabilities of the language as envisioned by the U.S. Air Force and the Institute of Electrical and Electronics Engineers (IEEE)*. The system will ultimately contain the tools for simulation, timing analysis, microcode retargeting and both silicon formulation and other forms of hardware synthesis.

Several research efforts were conducted concurrently with the one reported here. They are: 1) a "VHDL Language Analyzer," by Captain Deborah J. Frauenfelder whose purpose was to design and produce a prototype VHDL Language Analyzer which will provide the capability required to function as a front end processor for a hardware simulator in the design and development of VHSIC class chips (5) (See Block 1 in Figure 1.1), and 2) a "VHDL Hardware Simulator Using Parallel Processors," by Captain Michael S. Kamrowski whose purpose was to design and implement a simulator which runs on a parallel hardware simulator (6). It is shown as Block 5 in Figure 1-1.

---

* At this writing, IEEE is attempting to standardize VHDL.

1.3 Summary of Current Knowledge

The technological development of Hardware Description Languages began as early as 1939, when Shannon used a form of what is now considered hardware description language while working on switching circuits (7). The use of HDL's is not uncommon, as Lipovski noted in a special IEEE issue on HDL's. He observed that whenever someone developed a circuit simulator they felt apparently compelled to also develop a HDL to drive it rather than learn and adapt an existing HDL to their application need (8). Many prominent researchers have called for the development of a general purpose HDL (7). The problem has been that although many HDL's were adequate for specific purposes, none were entirely satisfactory over the range required for a large hardware design project.

In the early 1970's, the DOD directed the development of the Ada programming language in an effort to incorporate the features of modern high level programming languages and software engineering concepts such as structured programming, information hiding, data abstraction and handling, and real time control. This effort resulted in the designation of Ada as the standard DOD high order language (9). Similarly, while analyzing the problem of how to concisely communicate design infomation on integrated circuits containing upwards of 250,000 logic gates, the

VHSIC program office decided that the basic concepts and constructs implemented in Ada could also support a new HDL. Now in its final development, VHDL uses Ada as a guide, and incorporates the VHSIC specific requirements of portability, maintainability, timing, and the ability to hierarchically model and simulate integrated circuits. Generally, VHDL constructs supported by Ada are required to use Ada syntax. A detailed discussion of the interrelationships between VHDL and Ada can be found in the VHDL Design Analysis and Justification Report (10).

1.4 Problem Statement

A need exists for a VHDL Simulator to be an integral component of the AFIT VHDL Environment. As an important first step in creating a full simulation capability for the AVE, a prototype simulator kernel will be designed and implemented. The function of the kernel simulator will be to initially assist in the development and analysis of the VHSIC Hardware Description Language's suitability and effectiveness in modeling integrated circuits of the VHSIC class. Upon completion of the development phase of VHDL, the fully implemented simulator would permit the designer to check the validity of a VHDL description for a circuit design.

1.5 Scope

The purpose of this research effort is to study, model and create a prototype simulator kernel for VHDL in a UNIX environment. The interface required between the intermediate form VIA FILE and the simulator will be specified but not designed or implemented. The prototype simulator will not support the full capabilities of VHDL; however, a follow-on research effort is expected to develop the translator (preprocessor) which will set up the simulator from a VIA FILE and complete the development of the simulator itself. The detailed design of the prototype simulator presented here includes an installation guide, users manual, design description, and source code for that part of the design which has been implemented.

1.6 Approach

This research effort was conducted in three phases. Phase One consisted of a literature review of available material on hardware description languages and simulators. Phase Two required learning the syntax of VHDL and its constructs and capabilities. A thorough understanding of VHDL was essential for the design and implementation of the VHDL Simulator. Phase Three, comprising the majority of the effort, consisted of defining the requirements for the

simulator and designing and implementing the prototype simulator.

1.7 Assumptions

The VHDL Simulator is designed to run on a computer with a UNIX Operating System. For this reason, the C programming language was chosen to implement the simulator. A UNIX-based VHDL system may assist in overcoming some of the inefficiencies inherent in an Ada-based implementation.

1.8 Sequence of Presentation

Chapter Two gives specific requirements for the UNIX-based VHDL simulator. The general simulator requirements are presented, the operating environment is established, alternative approaches of implementing the simulator are discussed, and the tradeoffs inherent in each of the approaches are examined.

Chapter Three presents the general system design and explains the basic algorithm of simulation.

Chapter Four describes the detailed design of the simulator. A detailed description of the simulator structures, program modules, input parameters and output report is presented.

Chapter Five presents the results produced from exercising the simulator with test stimuli and provides an analysis of the results.

In Chapter Six the conclusions gained from this research are presented and recommendations for further research and development are offered.

II. Requirements Definition

2.1 General

The purpose of this chapter is to present the project objectives and the functional, implementation and performance requirements for a prototype VHDL simulator. The general requirements for a fully implemented simulator are presented first followed by the specific requirements for the prototype simulator developed which is the objective of this thesis.

2.2 Requirements for a Fully Implemented Simulator

2.2.1 Scope. The complete simulator must be able to simulate the entire scope of VHDL. The simulator must read a file of test data, and record some or all of the signal values generated during the simulation. The simulator will obtain the simulation control information, design description, test data and test setup from the VHDL Intermediate Access (VIA) format.

2.2.2 Compatibility. The Simulator must be compatible with VIA, the intermediate format currently being designed

for use with the AFIT UNIX VHDL system. Since the simulator is being designed and developed to operate in the UNIX Environment, the C language is the language of choice. There should be no difficulty in converting from VIA to C since the mapping is largely one to one. The translation from VIA to C is more easily achieved than with a strongly-typed programming language due to C's flexibility. C routines are created from VIA constructs and combined with a simulator run-time library. After compilation by the C compiler, the resulting simulator is executed.

2.2.3 Flexibility. The simulation program will require the flexibility to adapt to the differing designs of individual users and to adapt to changing technologies. The flexibility required cannot be satisfied by reprogramming. For this reason the hardware logic primitive constructs are provided through behavioral functions and architectures from VIA (11:2-5).

2.2.4 Input Test Vector File. The simulator must provide the user with the ability to describe the test data streams necessary to stimulate the design under simulation and verify that the outputs produced are correct. The test vector file should allow the user to specify the time, input data stream and variables that are to be associated with the

signals during the simulation. The user should also be allowed to define test data initialization values.

2.2.5 Interactive Capability. The user should be provided the capability to provide interactive input to the simulation. This should include the ability to describe test data, establish breakpoints, select the simulation time period, and specify the content and format of test reports. This interactive capability should be both by input file and from the command line.

2.2.5.1 Breakpoint. The breakpoint function allows the user to specify the events or the frequency when a breakpoint is to occur. When a breakpoint is specified, the simulator must output to a file the required data and then restart under user control the simulator operation, at the point of interruption, as if no stoppage had occurred.

2.2.5.2 Time Selection. The user should be able to define the time period over which the simulation is to be run relative to the start of the simulation. If the user does not designate the beginning and/or ending time period, then the default values will be used. The user should also be able to specify the actual time units used, e.g., nanoseconds, microseconds, etc.

2.2.5.3 Content and Format of Test Reports. The user should be allowed to identify by name the signals wanted in the output report. Additionally, the user should be able to define the type of signal trace desired. Examples of types of signal trace which may be desired are:

1. A sampling of all selected signals at predefined time intervals.
2. A trace that displays ony events.
3. A trace that displays all transactions.

2.2.6 Multiple Drivers. The simulator must be capable of processing signals that have multiple drivers. In VHDL a hardware network is modeled with a signal that has an associated bus resolution function. This type of signal is called a bus. The bus resolution function, which is user defined, provides a procedure for resolving the values of the signal's multiple drivers into a single value. The bus resolution function takes an unconstrained array as its input and returns a single value of the same type as its output (12:6-5).

2.2.7 Delay. VHDL allows the designer to model either inertial or transport delay. The simulator, therefore, must be capable of processing either inertial or transport delay. The specific delay is executed when a set of transactions is being used to update the projected output waveform of the

current and future values for a driver. For transport delay, the reserved word transport will appear on the right hand side of the signal assignment statement, otherwise inertial delay is the default (13:8-5).

2.2.8 Networks. Signals that are associated with each other by a port association list form a network. Because of these association lists on block statements (and through component instantiation statements), the value of a signal cannot be determined independently of the values of the signals associated with it. The simulator needs to recompute the value of the network associated with a driver, whenever the future value of a driver in the network becomes the current value of the driver (12:9-1).

2.2.9 Abstraction Capability. The simulator must be capable of allowing the processing of packages. This will enable the simulator to handle hardware devices described at higher levels of abstraction. The packages are used to group together related declarations which may include user-defined types and subprograms. The user-defined types allow the designer to add to the predefined language types and then use subprograms to permit operations on these data types (12:16-1). Packages also permit designers to share the data stored within the packages.

2.2.10 Steady State. The simulator must monitor its operation to detect the following three possible conditions:

1. No transactions remain to be processed and no transformers are active.
2. The number of transactions are at a static level and are not decreasing.
3. e transactions in process have exceeded an ected upper limit for the signals being cocessed.

Simulator oscillation control is required to detect and correct this simulator oscillation. Oscillation control consists of identifying the oscillation, eliminating the oscillation and finally continuing the simulation process (14:242).

2.3 Requirements for the Prototype Simulator

2.3.1 Objective. Although an entire VHDL simulator is the ideal goal of this research, time only permits a small subset to be created. Thus, the primary goal of the research reported here is to create a prototype simulator kernel for VHDL that will operate in a UNIX environment. The program will implement the major basic functions of VHDL using the C programming language. The prototype simulator will provide proof of the design concept.

To satisfy this goal, the program will be designed to meet the following objectives:

1. Minimize the memory required for the executable simulation program.

2. Maximize simulator execution speed.

3. Provide the user with flexibility in the choice of naming the input and output files and the ability to specify the simulation start and termination times.

4. Provide detailed error checking and clear and concise error warnings and messages.

The requirements for these objectives are specified in the following section.

2.3.2 Requirements Definition. This section defines the requirements for the prototype simulator. First a description of the program and how it is to be used is provided. This is followed by the three categories of program requirements - functional, implementation, and performance requirements.

2.3.2.1 Program Description. The prototype simulator program, referred to as VSIM (VHDL Simulator), reads an input file of test vectors and evaluates the transformers to determine if an event has occurred. If an event has occurred on a signal, then the fanout list for the signal is evaluated and the related behavioral functions are executed to produce new transactions. An output trace file of all events that occur during the simulation is produced.

VSIM performs the following major actions:

1. It interprets the command line and establishes the runtime environment necessary to execute the requested user options.

2. The internal data structures are created and initialized from the designated input data files. In the fully implemented simulator, these data files are automatically created from the full VIA descriptions.

3. The input vector file is read, and time is incremented on the simulator clock.

4. Events are evaluated and new transactions created.

5. An output file is created.

6. Error checking occurs continuously throughout the other major processes. The command line and the input vector file are also checked for errors and all errors are displayed on either the user's terminal or written to an output file.

2.3.2.2 Functional Requirements

2.3.2.2.1 Approach. The two basic classes of simulator are compiler-driven and table-driven event-directed. Most of the modern simulators are table-driven event-directed since this type is more versatile in handling delays and also reduces the required simulation time (14:203).

VSIM will use an approach similar to the one being implemented by Intermetrics in the VHDL Build 2 Simulator (11:4-40). This approach involves implementation of a precompiled simulator instead of an interpretive simulator

program. The VSIM program will be a compiler-driven event-directed simulation. It differs from the Intermetrics approach in the data structures used and the programming language (C versus Ada).

2.3.2.2.2 VSIM Created Structures and Functions. For the prototype, the data structures and the behavioral functions that drive the prototype simulator kernel will be created internal to the program since the present state of VIA is not developed sufficiently to provide these functions for the prototype simulator. The assumption was made that the structures and functions created for VSIM would be directly related to the structures and functions that VIA will provide when completely defined. The structures and functions also must appear as if they are directly traceable to VIA.

The VSIM prototype must read the data necessary to establish the structures and access the behavioral functions from the following three files: **signal data, driver data,** and **fanout data** in order to prove the design concept. VSIM must access the functions as if they were traceable to VIA. To accomplish this, the addresses of the behavioral functions must be stored, and the functions called at execution time by reference to these addresses.

2.3.2.2.3 Simple Signal Assignment Statements. VSIM must be capable of executing the operations required by a simple signal assignment statement. This requires that the program 1) determine the name of the signal that will receive the output of the assignment, 2) post the new future value of the signal, and 3) schedule the value to occur on the signal after the specified time delay. In posting the new value, VSIM will need to determine the appropriate delay (transport or inertial) and schedule the value to occur by correctly placing it in an ordered list and deleting old transactions depending upon the delay type.

2.3.2.2.4 Blocks. VSIM will handle single block statements. Guard lists as part of networks will not be implemented in the prototype.

2.3.2.2.5 Processes. VSIM implements the process statement as a simple signal assignment statement. VSIM does not model the sensitivity list or declarative parts of the process statement. VSIM does implement independent processes that can execute a sequence of statements and schedule new events to occur.

2.3.2.2.6 Data Types. The VSIM data structure, to be implemented as a union, will support only integer and floating point data types. VSIM allows for four

input values: '0,' '1,' 'z' (high impedance), and 'u' (unknown state). VSIM will not implement packages, subprograms and user-defined types.

2.3.2.2.7 Single Drivers. VSIM assumes that each signal has at most one driver. The driver data structure will support multiple drivers per signal; however, the program controller will not support operations on multiple drivers such as the bus resolution function.

2.3.2.2.8 Input Test Vector File. VSIM will read and interpret the input vector file. The user specifies the input filename on the command line. Failure to specify the input file will result in a fatal error and termination of program execution. VSIM will only read the time and associated signal input vectors; it will not accept the assignment of simulator variable values. The $ will be used as a delimiter between the designation of the input ports (signals) and the associated input vectors.

2.3.2.2.9 Interactive Capability. VSIM will allow the user the selection of several options through the command line. At a minimum, the command line must contain the program name VSIM and the name of the input vector file. The following format must be used for the command line:

VSIM [options] input file name

The command line options may be specified in any order and the program is insensitive to the number of allowable arguments which are present. The command line options are summarized below:

-d n Selects the debug option which causes the contents of selected structures to be printed to the output file. Debug prints the selected data the first time that simulate is entered and each time a transaction is processed. n can be one of four integer values (1, 2, 3, 4) where:

n = 1 prints all signal structures

n = 2 prints the event queue

n = 3 prints all transaction queues

n = 4 prints all data structures

-o filename Allows the user to select the output file where the output trace file will be written. The default is **sim output.**

-s n Allows the user to select the simulation start time. The time units specified must be the same as the units of time used for the input vector file and the circuit description. n must be an integer value. The default start time is 0.

-t n Allows the user to select the simulation termination time. The time units specified must be the same as the units of time used for the input vector

file and the circuit description. n must be an integer value. The default termination time is 10000.

-b n Allows the user to select a breakpoint time when data will be dumped to the output file. Simulation continues after the breakpoint is processed. n must be an integer value.

2.3.2.2.10 Timing. VSIM will allow the user to select the simulation start or termination time as indicated above. Either one or both may be selected; however, a fatal error occurs if the termination time specified is less than the start time. Simulation time is not discrete but is event-directed, i.e., it is incremented according to the scheduled occurrence of events.

2.3.2.2.11 Error Checking. VSIM will provide extensive error checking. If errors occur, they are displayed on either the standard output or written to the specified output file. An error on the command line or a failure to open the user specified input vector file will result in a fatal error message displayed on the standard output. All other error messages are written to the specified output file. Errors can affect program execution in one of two ways: 1) After warnings program execution resumes, or 2) after fatal errors program execution is aborted and no output file is written.

2.3.2.2.12 Initialization. Prior to the start of simulation, all the signal and driver values will be initialized to 'u', the unknown state.

2.3.2.3 Implementation Requirements

2.3.2.3.1 The UNIX Environment. The prototype simulator will be designed to operate in the UNIX BSD 4.2 Environment.

2.3.2.3.2 The C Programming Language. The program will be written in the C programming language. The C language was chosen because:

1. It is the primary language of choice for applications in the UNIX environment.
2. The flexibility of C provides an advantage over a more strongly typed language.
3. There is no difficulty in translating from VIA to C; in fact, there may be a close to one-to-one mapping.

2.3.2.3.3 System. At a minimum, the simulator will execute on a DEC VAX.

2.3.2.4 Performance Requirements. Since the primary objective of VSIM is to simulate VHSIC class designs, the simulator must be very efficient in its use of memory and in execution speed. Figure 2-1 shows the

difference between the VHSIC class of problem and the less complex problems solved by Engineering Analysis Simulation Systems (11:4-39). The larger number of gates being simulated in VHSIC class problems (50K to 100K gates or more compared to 10K to 50K gates) usually consume vast amounts of memory on large minicomputers or mainframe computers and have very long execution times. The prolonged execution time and needs for a vast amount of memory program space can be unacceptable in a multiuser environment, particularly if the simulator program is not both memory and execution time efficient.

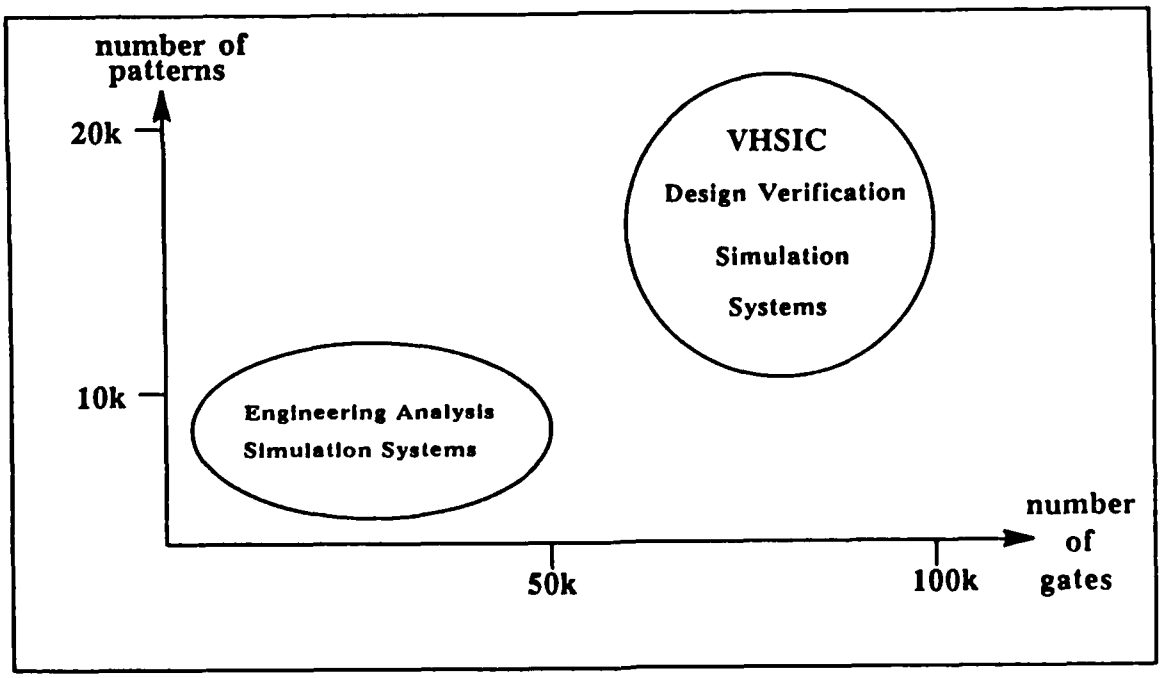


Figure 2-1 Engineering Analysis vs. Design Verification Systems (11)

The compiler-driven event-directed simulation approach used in VSIM is intended to improve the simulator's efficiency. Figure 2-2 shows the difference in efficiency between the interpretive and precompiled simulators. The interpretive simulator is usually more efficient to set up but less efficient as the number of test patterns increases. In VHSIC class designs the number of test patterns required for simulation is beyond the point where the interpretive simulator is more efficient than the precompiled simulator (11:4-40/41).

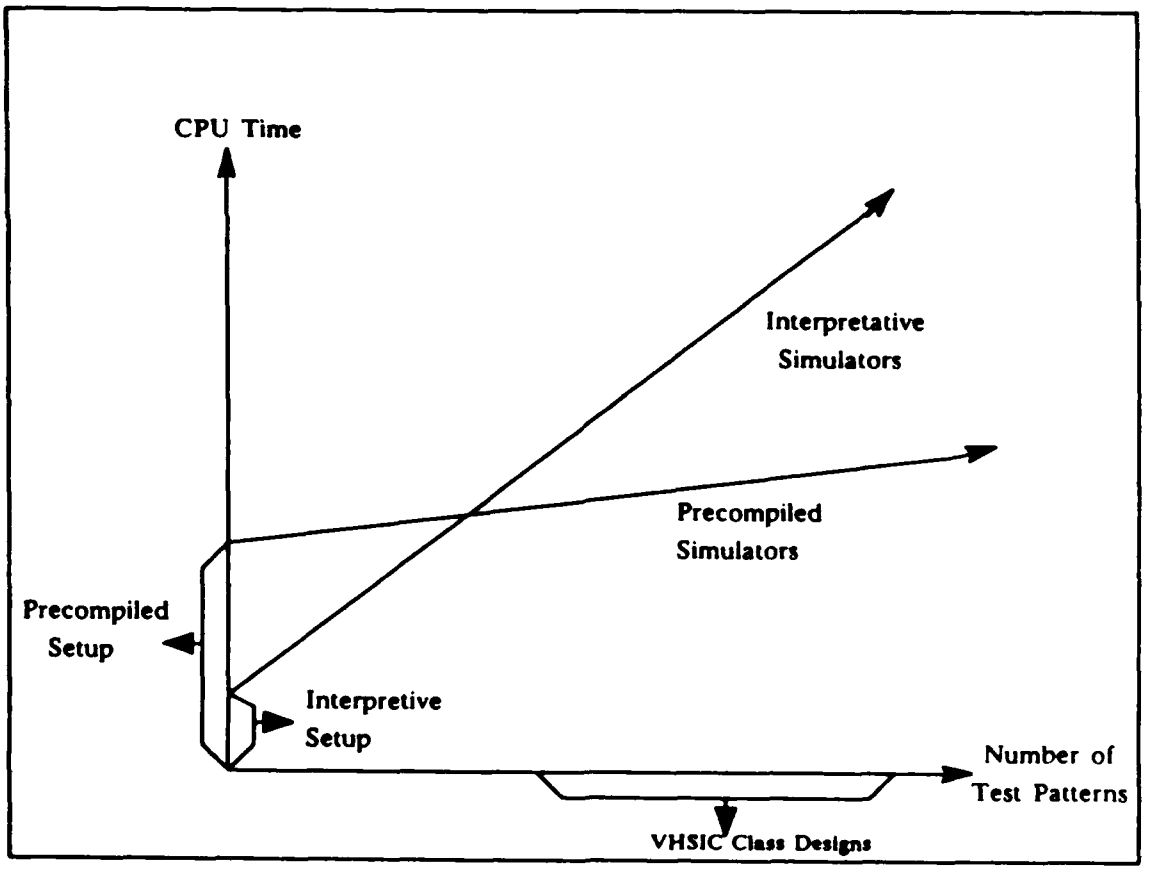


Figure 2-2 Precompilation Trade-Off Graph (11)

III. System Design

3.1 General

The system design is the overall plan which describes the general approach used to implement the program requirements. Since this thesis involves the development of a software program, existing software engineering tools have been used for the program's design and implementation.

An approach similar to the Structured Analysis and Design Technique (SADT) was used for the system design due to the author's familiarity with it and its effectiveness as a system level design tool. This chapter presents the system level design of VSIM using SADT-like diagrams.

3.2 System Overview

Figure 3-1 presents the top level system view of the inputs, outputs and controls of VSIM. VSIM reads and processes the user commands from the command line. This controls the optional program features and is used to tailor the output file as desired. VSIM reads the user specified input vector file and produces an output trace file. The data structures which VSIM uses during program execution are created based upon input received from VIA. Since the

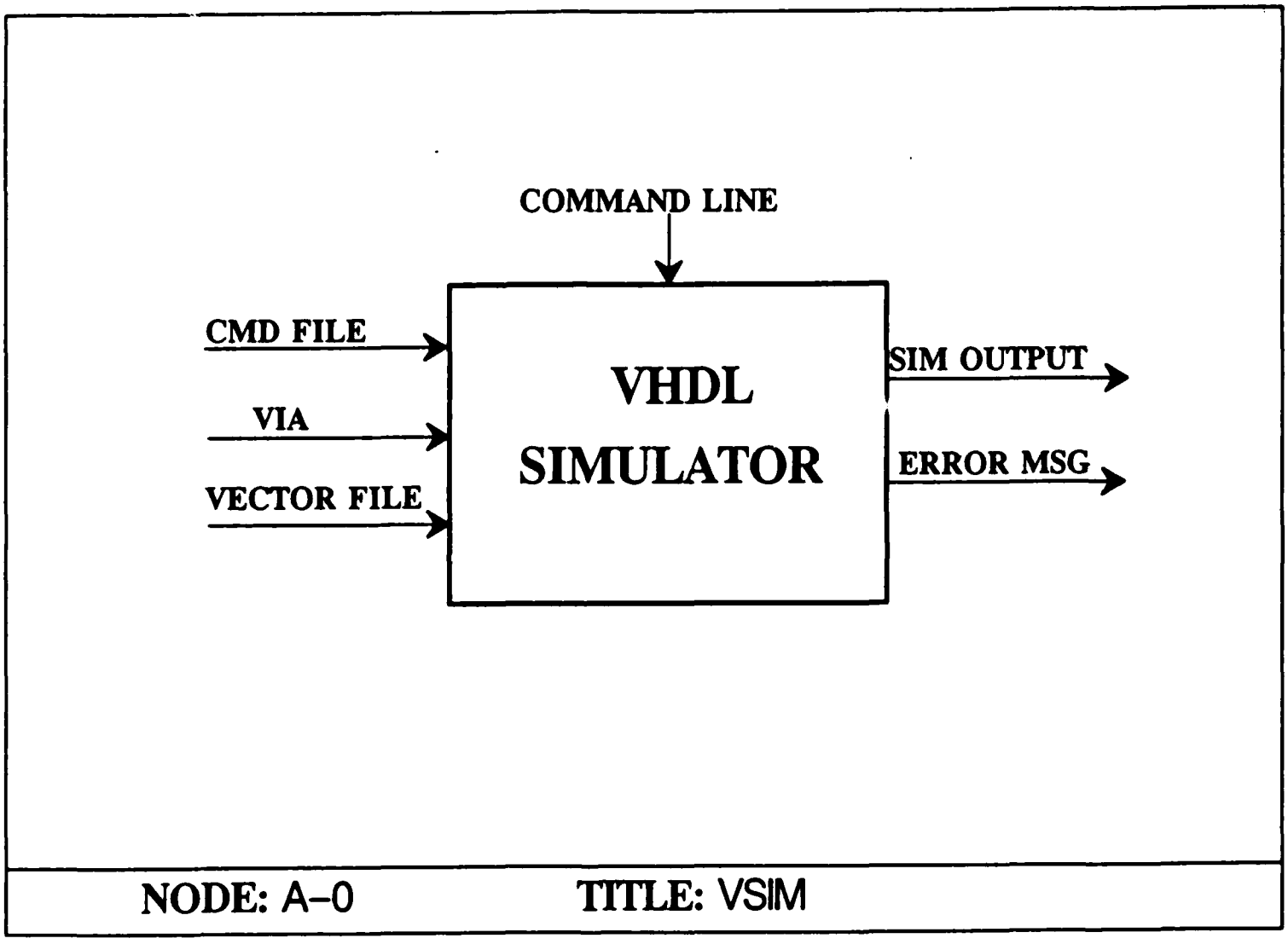


Figure 3 - 1 VHDL Simulator

present state of development of VIA is not sufficient to enable VSIM program execution, three input data files are used to drive the simulator. VSIM processes the input data files, establishes the internal simulation structures and then reads the input vector file. VSIM checks the input file for syntax and runtime errors. Incorrect syntax or runtime errors will result in the second type of program output: error messages to either the standard output or the specified output file. Errors made on the command line entry cause error messages to be written to the standard output. All other error messages are written to the specified output file. The type of error - warning or fatal - determines whether an output trace file will be produced. A fatal error terminates production of the output trace file.

VSIM is designed to also read user input from a command file; however, this feature will not be implemented in the first prototype version.

The method used in the remainder of the chapter will be to first provide a general description of the design of the major upper level program modules. This is then followed by a more detailed description of each of the upper level modules by decomposing them into their major subordinate functions.

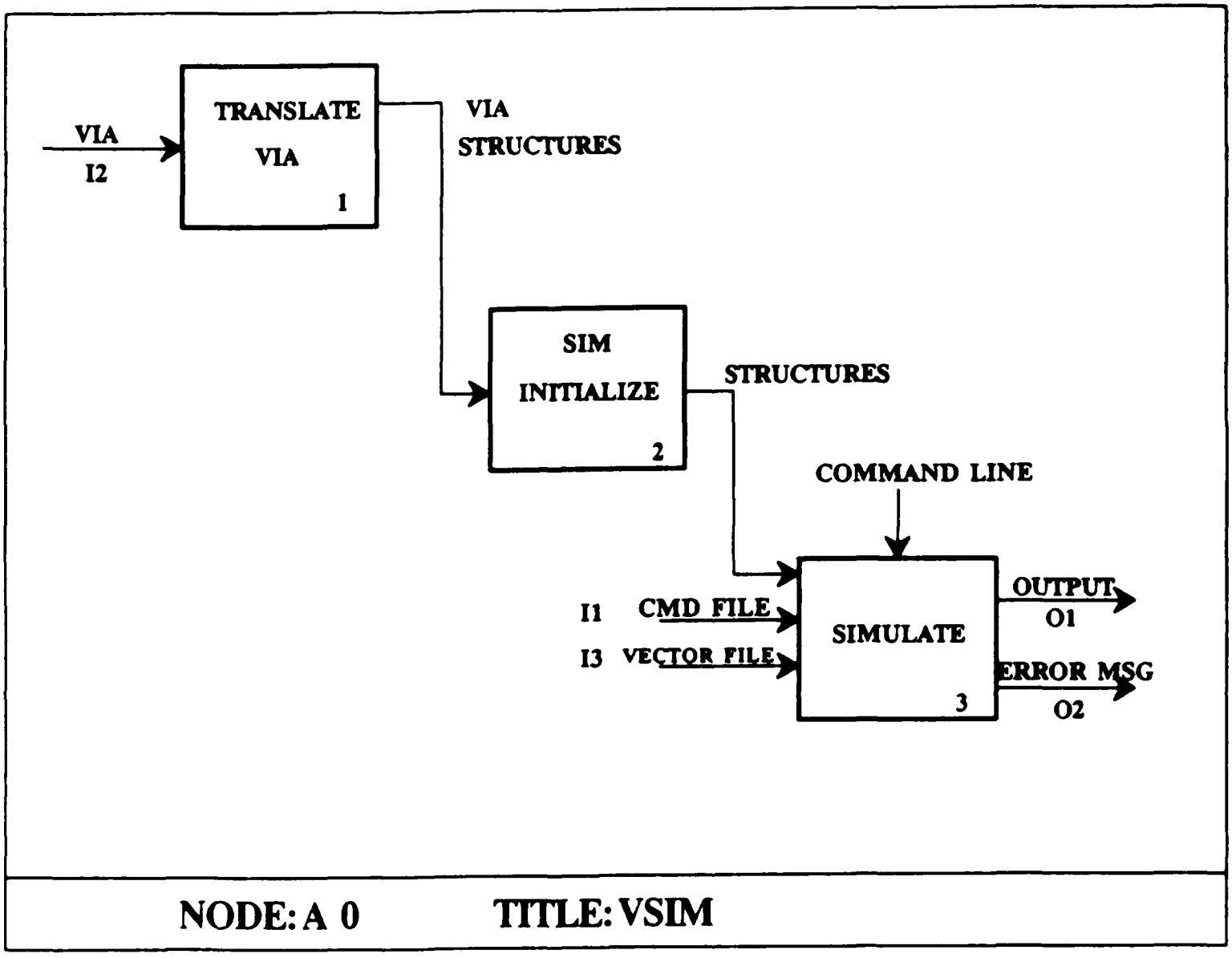


Figure 3 - 2 VSIM

program uses three data files within the **Sim Initialize** module - **Signal Data**, **Driver Data** and **Fanout Data*** - to perform the **Translate** VIA module functions.

3.3.3 **Simulate**. The **Simulate** module schedules and executes the simulation. User commands from the command line are read and interpreted to control the simulation. Initially, all signal values are set to "u" (Unknown) and time is initialized to the largest integer possible on the machine. When operating in the steady state, a simulation cycle begins when time is incremented to the time of the earliest scheduled transaction. This transaction is then evaluated to determine if an event has occurred, that is, if the value in the scheduled transaction is different from the current value in the parent signal. If there is an event then the new value is posted to the signal and the driver structures and propagated to all the outputs on the fanout list of the affected signals. The behavior functions associated with each affected driver are then executed producing new transactions which are posted to all applicable drivers in the network. The next scheduled transaction is then taken from the event list. If its time is different from current time, then the time is incremented

---

* The contents of these files are described in Appendices D, E, and F.

and the process is repeated. Each time a transaction is scheduled to become an event (i.e. it is now being processed to determine if there is an event) it is removed from the transaction queue. If the transaction was an input vector, then a flag is set and at the next time increment another input vector is read and posted. Figure 3-3 shows the simulation process.

The simulate module logically operates on two nested simulation cycles. The micro or inner cycle operates in delta time within the current simulation cycle.* All events scheduled for delta time are processed and all new transactions with a delta time are posted to the drivers of signals and executed in delta time. A delta cycle terminates when there are no more delta transactions to process. The concept of delta time in VHDL is used to insure the proper order of execution rather than as a meaningful time increment. The second cycle, the macro or outer cycle, consists of all transactions scheduled for other than current simulation time, i.e. some future time. Execution continues until all events have been processed or

---

* Current simulator time should be thought of as being divided into two different times - zero time and delta time. During zero time nothing progresses. Zero time is used for the simulator to do its own housekeeping. Delta time is an infinitesimally small increment of tangible time that is smaller than the time required to activate the lowest sensitivity. The summation of all the delta times within a current time cycle is less than one unit of time.

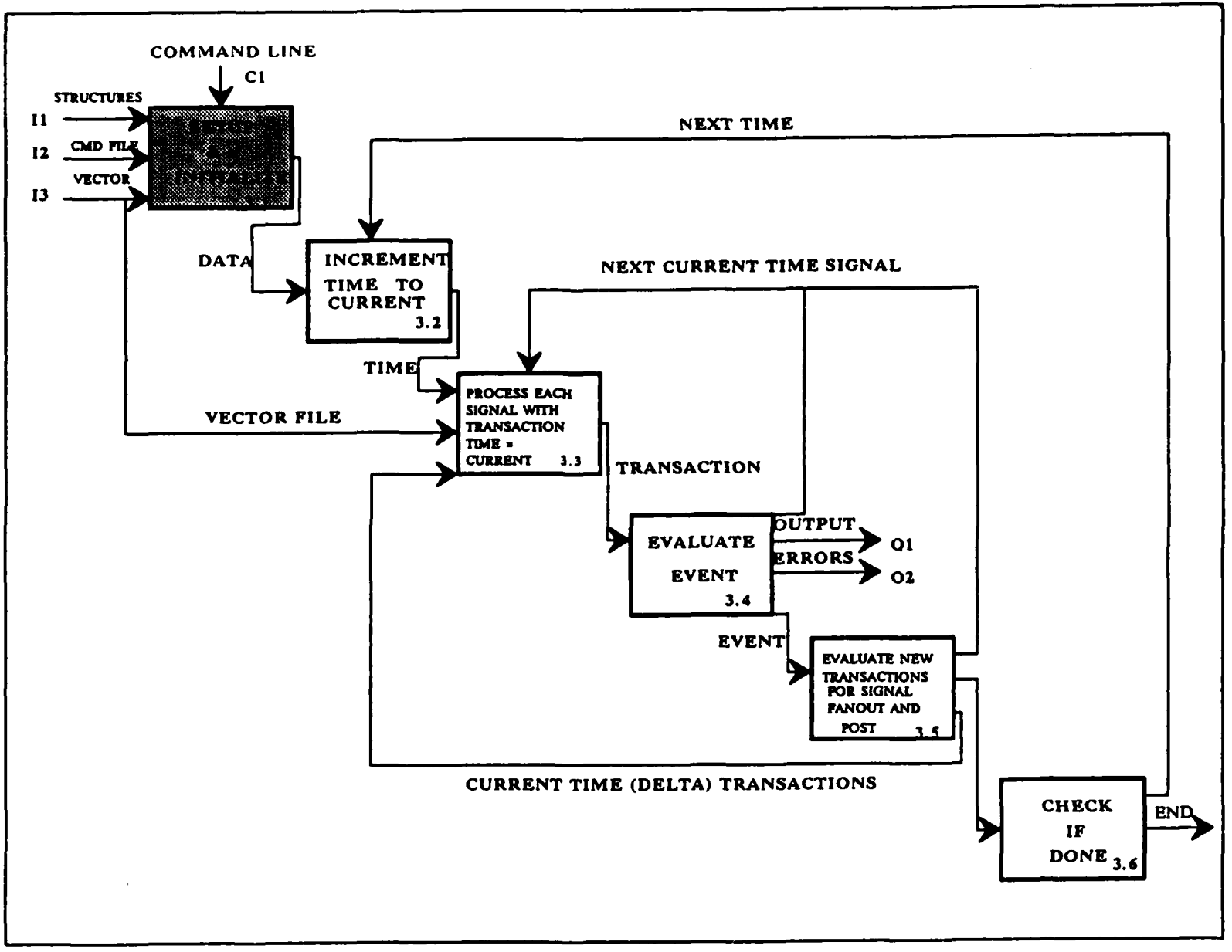


Figure 3 - 3 Simulation Process

a preset termination time is reached. Throughout the simulation cycle, specified data is written to the output file. Non-fatal error messages are printed to the specified output file. Fatal errors cause program termination.

These three modules and their function are further decomposed and described in greater detail in the following sections. The diagram associated with each module is included immediately following the description of the module and may be referred to while reading.

3.4 Translate VIA (Figure 3-4)

When implemented the **Translate** VIA module will create the data structures required to execute the simulation by traversing the Directed Acyclic Graphs (DAGs) which compose VIA. This module will translate the data in the DAGs into structures (block 1.2) that can be executed by the simulator. A precedence matrix which contains the fanout data for each signal will be created. The fanout data associates all signals in a given network and provides a pointer to the address for the corresponding behavioral function. Finally, the behavioral functions are translated from VIA into executable C code. The C language is used for the reasons discussed previously in Chapter 2.

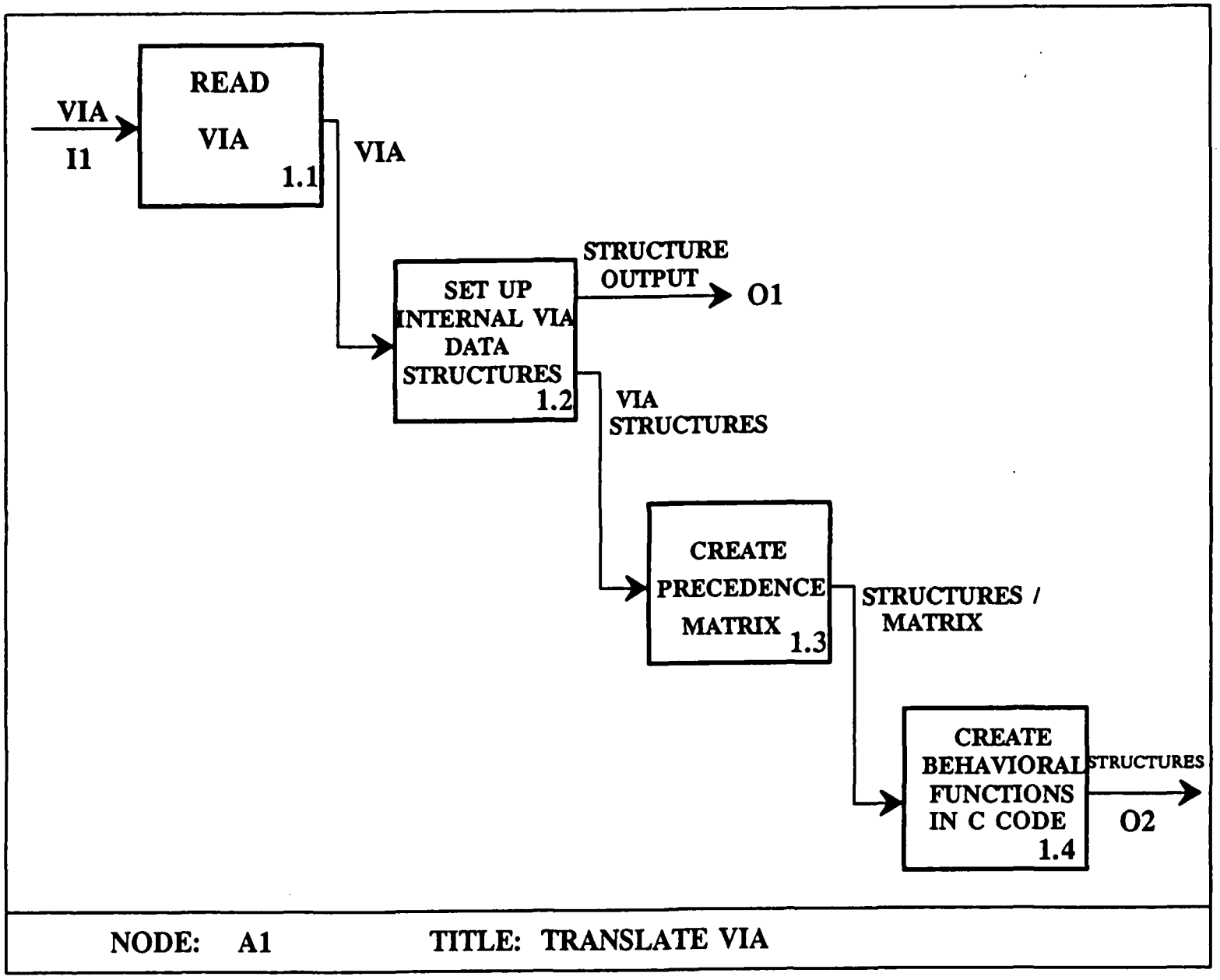


Figure 3 – 4 Translate VIA

3.5 Sim Initialize (Figure 3-5)

3.5.1 **Create Signals.** The **Create Signal** module controls the creation of the signal, driver and fanout structures. The three data files (**Signal Data, Driver Data** and **Fanout Data**) are opened and read until all the structures have been created initialized and linked. The data files are closed when all the structures have been created and the end-of-file is reached.

Figure 3-6 presents the data structures which are created. The transaction structure is created in the **Get Input Vectors** module but the structure is presented here for clarity and ease of understanding. Efficiency in both storage space and execution time was the primary consideration in determining the design of the data structures. This consideration for efficiency resulted in the decision to almost exclusively use dynamic data structures (linked lists) as opposed to static structures (arrays). The event queue linked list in the driver structure is designed as a doubly-linked list since it needs to be maintained as an ordered list which is constantly being updated and reordered as transactions are posted and deleted. The sort key field, which contains the time value of the first transaction in the driver's transaction queue in the driver structure, is used as the key to sort the event queue. All other linked lists are singly-linked

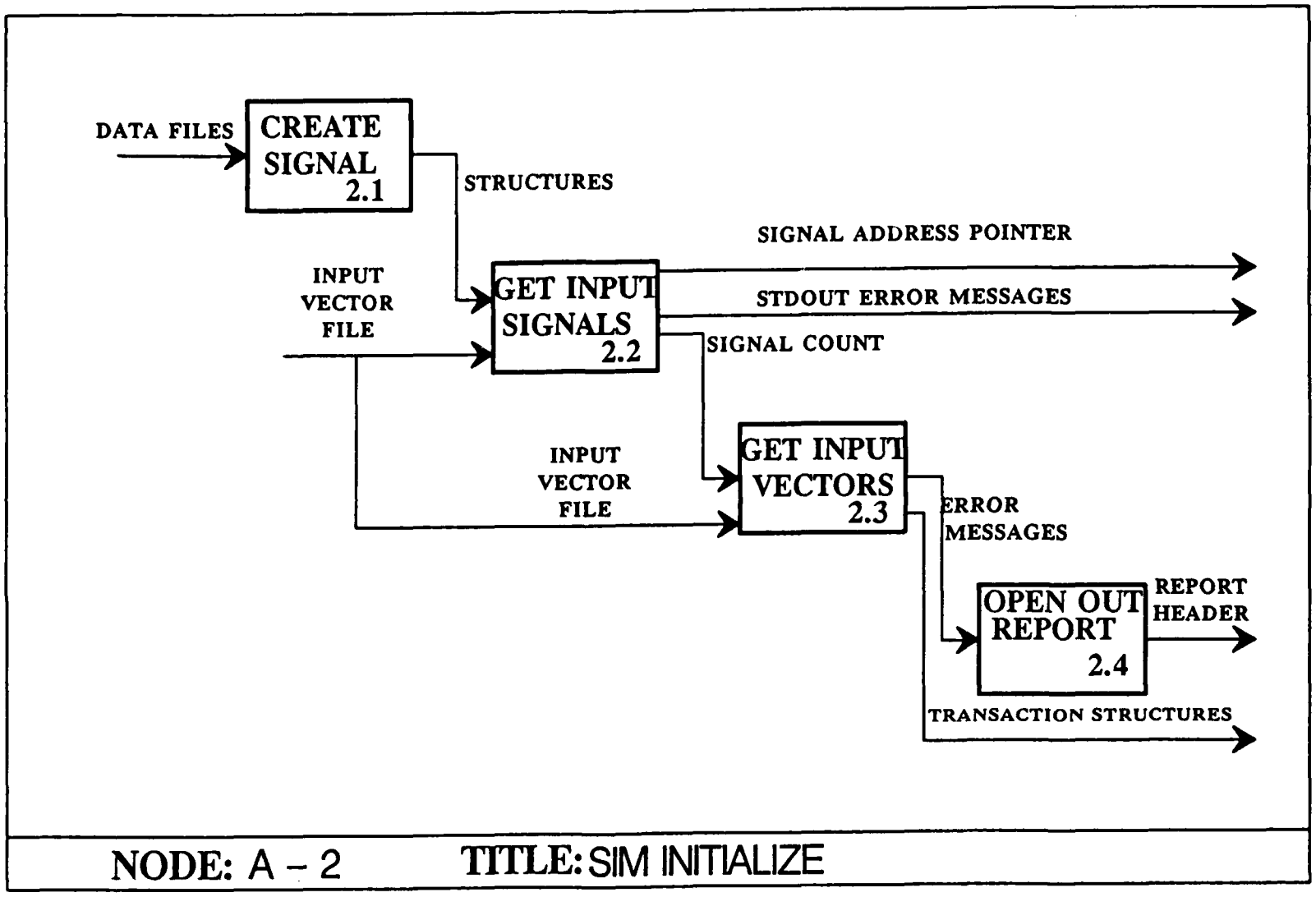


Figure 3 - 5 Sim Initialize

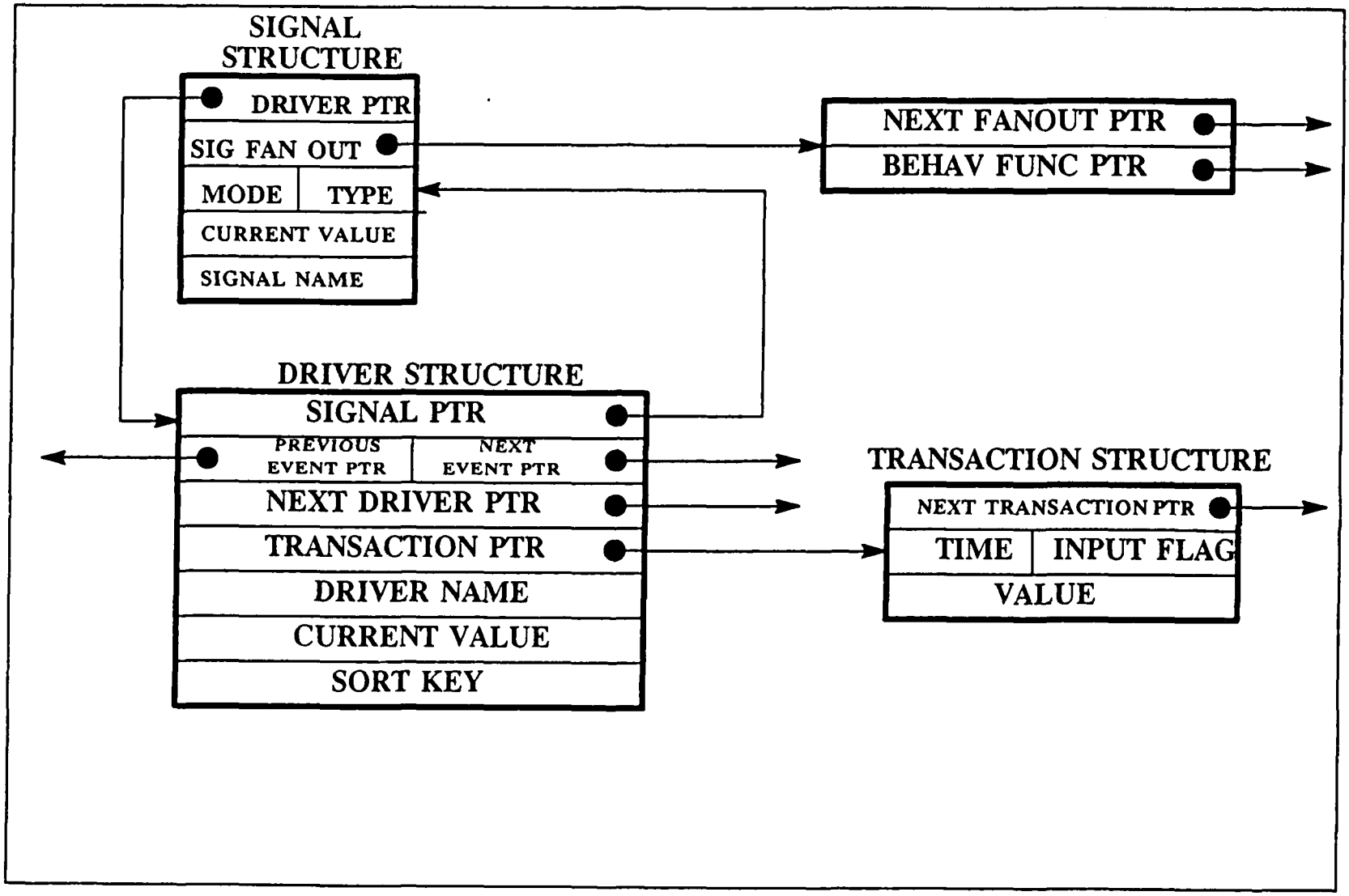


Figure 3 - 6 Data Structures

lists. This conserves storage space while providing adequate efficiency. The signal and driver name fields are implemeted as unconstrained arrays to support the VHDL convention of placing no limit on name length.

The structures are connected as follows:

1. Each signal structure is linked to the first fanout structure in the fanout list. Each succeeding fanout structure is linked to the preceding one.
2. Each signal structure is linked to its subordinate driver. In the case of multiple drivers, the signal structure is linked to the first driver and each succeeding driver is linked to the preceding one.
3. Each driver is linked to its parent signal structure. All drivers are linked together in the doubly linked list which functions as the event queue.
4. Each driver is linked to the first transaction structure in the transaction queue. Each succeeding transaction structure is linked to the preceding one.

3.5.2 **Get Input Signals**. The **Get Input Signals** module opens the user specified input vector file and reads the input signal names from the file. A failure to open the specified input vector file results in a fatal error and program termination. **Get Input Signals** reads the input file creating an array of pointers (**sig array ptr**) which stores the addresses of the input signal structures. These addresses are subsequently used throughout program execution for the posting of input vectors. A control variable, **Vector Count**, is initialized to specify the format for

reading the input vectors from the file. **Get Input Signals** reads until it encounters the $ delimiter.

3.5.3 **Get Input Vectors**. **Get Input Vectors** reads the input vectors from the input file using the control variable **Vector Control**, parsing the input and converting the string characters to integers. When a valid input vector and its associated time has been processed a transaction structure is created, posted to the appropriate driver, and the event queue updated. All transaction structures created by **Get Input Vectors** are assigned an input vector flag of 1 to distinguish them from transactions created during the simulation cycle. An invalid input vector character is ignored and results in a warning error message.

3.5.4 **Input Vector File**. Figure 3-7 presents the format for the **Input Vector File** and Appendix C contains an example of an **Input Vector File**. The $ delimiter separates the input signals from the input vectors. There is no limit on the number of signals that the file may contain. However, for each input signal there must be a column in the input vector section. The time of type integer is the first value in each input vector row. The input vectors must be monotonically increasing, and no two rows of input vectors can have the same time value. Following the time are the input vector values corresponding to each signal. The only

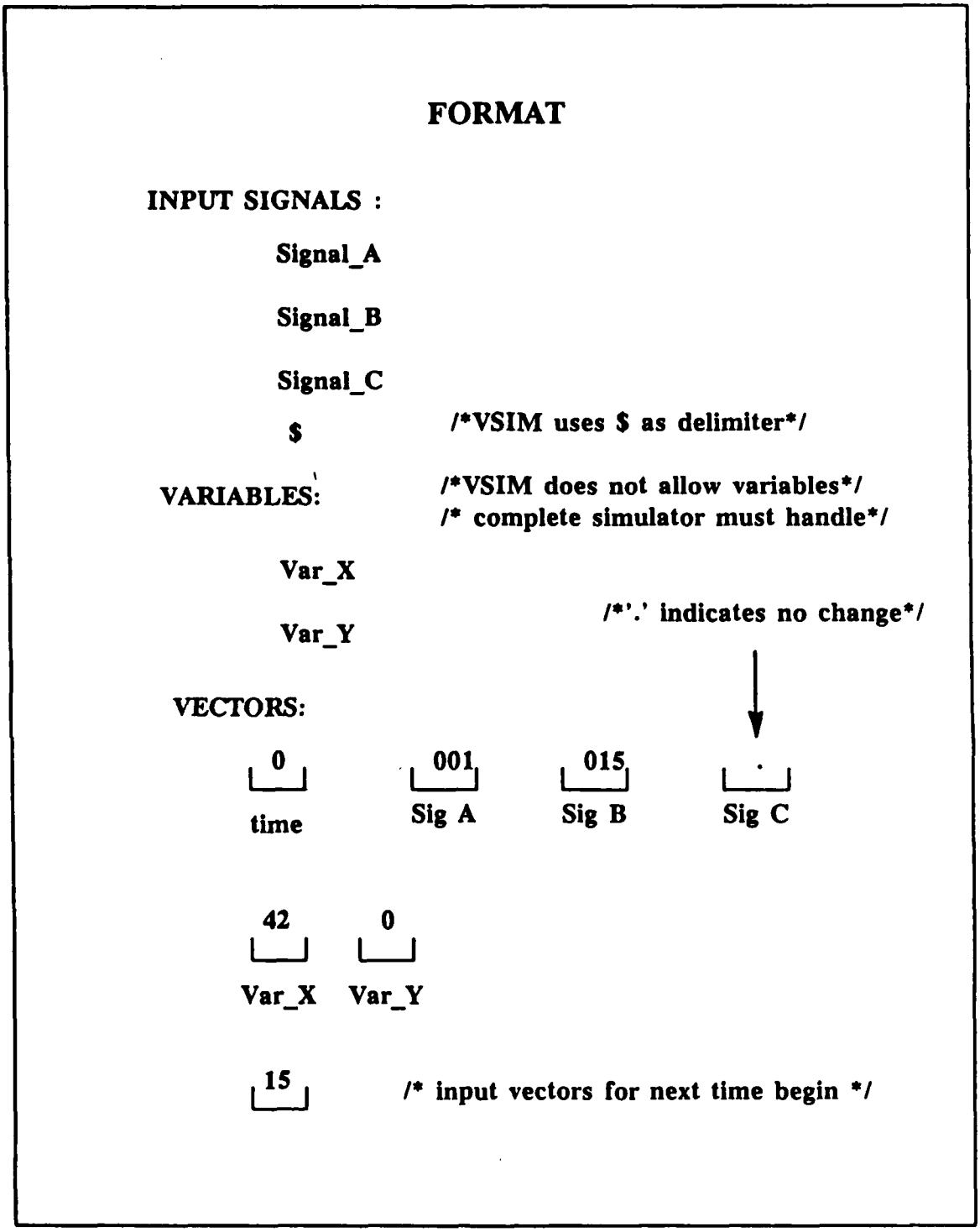


Figure 3-7 Sample Input Vector File

valid characters are '0', '1', 'z', 'u' and '.'. The '.' indicates that the previous vector value for the signal has not changed. Each value on the input vector line must be separated by a white space character.

3.5.5 **Open Out Report**. The **Open Out Report** module opens the output report to the user specified file or the default and prints the output report header.

3.6 Simulate (Figure 3-8)

3.6.1 **Sim Clock**. The **Sim Clock** module increments the simulation clock when the next scheduled event has a time value greater than current time. **Sim Clock** sets a control variable to cause program termination when the maximum simulation time is reached.

3.6.2 **Pop Trans**. The **Pop Trans** module deletes a transaction and frees the memory space. The event queue is updated after each deletion.

3.6.3 **Process Fanout**. The **Process Fanout** module accepts a signal which has had an event and evaluates the signal fanout for the event. The value of the event is propagated to all the outputs on the fanout list and the behavior functions are evaluated. New transactions are

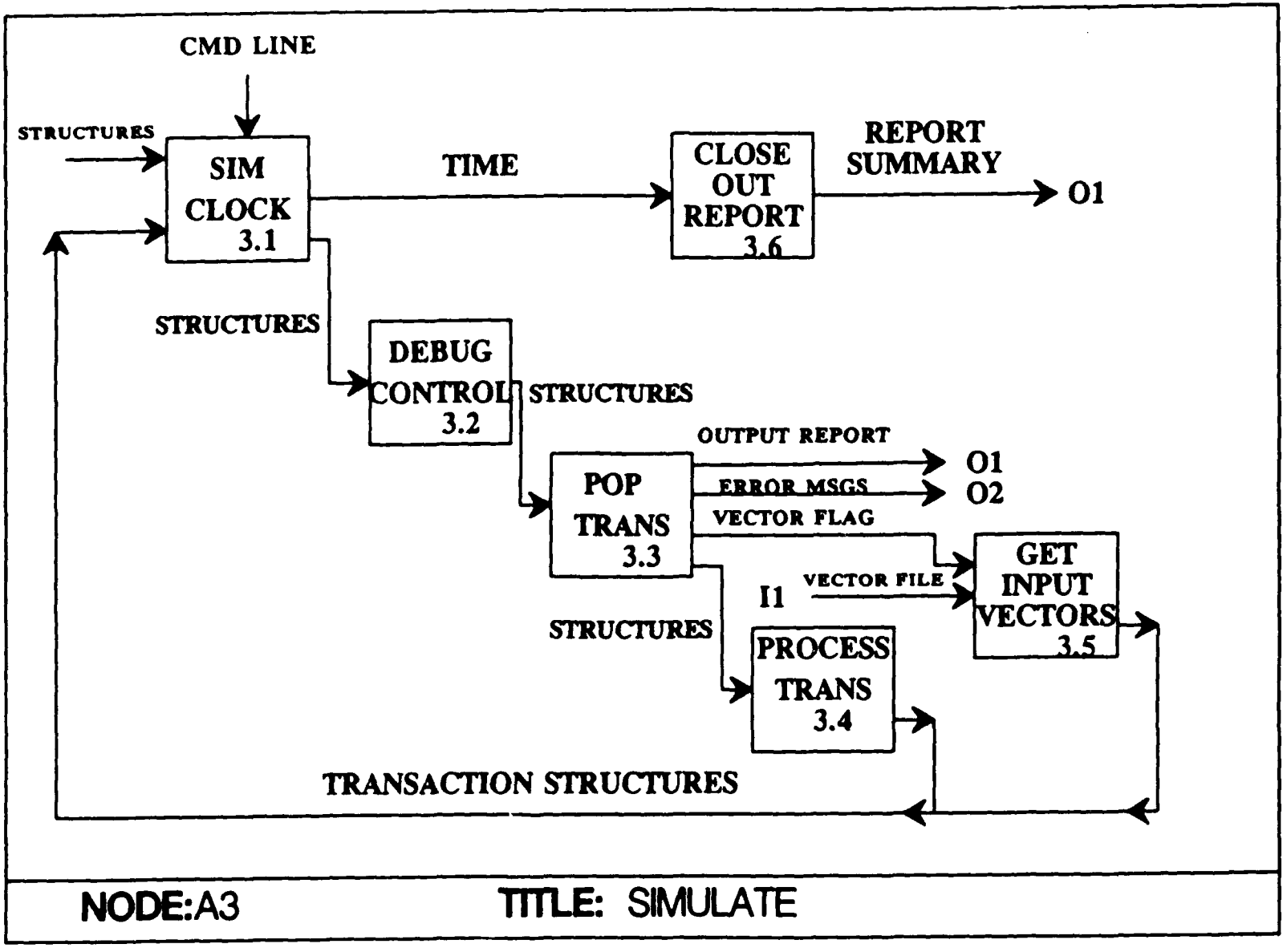


Figure 3 - 8 Simulate

posted in either delta or future time to the target signal's driver and the event queue is updated. Evaluation of the fanout list continues until all behaviors on the fanout list have been executed.

3.6.4 **Get Input Vectors**. The Get Input Vectors process is executed after an input vector (a transaction with an in vector flag of 1) has been processed in current time, prior to incrementing simulation time. The next input vector line is read, transaction structure created, posted to the appropriate driver and the event queue updated.

3.6.5 **Convert**. The Convert module parses the output values and converts the integer values to characters for output. An error message is output if an invalid character is encountered.

3.6.6 **Close Out Report**. The Close Out Report module formats and outputs the summary for the output report when the simulation is terminated. The total simulation time is calculated and the output report file closed.

IV. Detailed Design

4.1 General

The purpose of this chapter is to present the detailed design of VSIM. A brief discussion of the design goals and design procedures is presented first. This is followed by a presentation of the detailed design, which is the major focus of the chapter. The detailed design of the major program modules is presented using hierarchical charts and structured English. Structured English was chosen as the design specification tool because of its understandability, simplicity and conciseness. Additional information is provided in those specific instances where a more detailed explanation is required for a complete understanding of a module's function.

The detailed design of translating the general specifications documented in the previous chapter (System Design) into a comprehensive plan for implementing VSIM was specified sufficiently to minimize problems during the program implementation phase. A significant amount of the implementation time expended on VSIM occurred during the detailed design period. This methodology proved to be very beneficial during the coding and implementation phase as

there were no major design errors encountered and the debugging effort required was less than expected.

4.2 Design Goals

In addition to the major design goals stated in the requirements section of Chapter II, the following are the specific goals for the translation of the requirements into code. The first goal was to use the generally accepted good software engineering practice of having loosely coupled modules with good cohesion. The code must be easy to understand and avoid workable but confusing program constructs. Separate procedures should be used to implement a single and specific, possibly repetitive function, i.e. the program should be modular. Variable and function names should be chosen which are descriptive of the data they contain or the function that they perform.

The above goals were satisfied in the implementation of the detailed design of VSIM. The program is highly modularized, and, as much as possible each function was designed to perform single or highly related functions. Variable and function names were carefully chosen to be representative of the data represented or the function performed.

As discussed previously, since memory space and execution time are very limited relative to the requirements of a VHDL simulation, the efficiency of the program is critical. Program space for storing data must be minimized whenever possible. Proper implementation of these goals should produce a program which is easy to understand, maintain and revise.

4.3 Design Procedure

The first action in the detailed design process was to begin converting the SADT-like diagram specifications into actual descriptions of functions which could be implemented in the C language. Generally, the SADT diagram specifications were directly translated into the corresponding high-level C function description. This high-level function description usually required several actual C functions to implement the specific requirements of the high level specifications. Of course, not all of the SADT descriptions were mapped one-to-one to code. Some had to be modified during the coding process. In other cases, a better method of implementing a particular process or function was developed and the previous design details were changed to reflect the improvement.

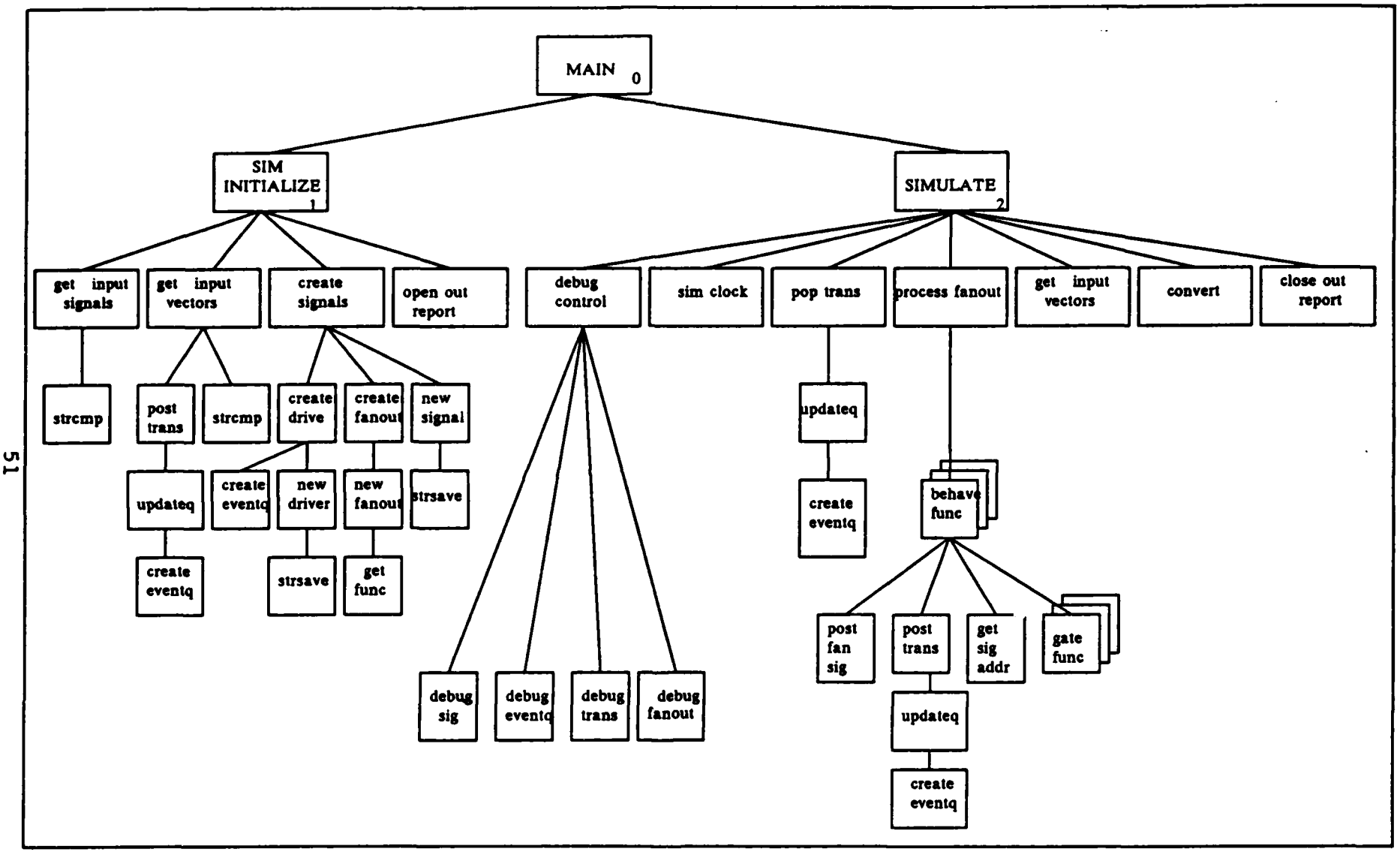


Figure 4 – 1 VSIM Hierarchy Chart

4.4 Function Descriptions

The detailed design presented here follows the basic structure presented in Chapter III, System Design. The important major modules are presented along with corresponding hierarchical charts and structured English descriptions. Figure 4-1 shows the hierarchical structure of VSIM to provide a system level description of the program.

4.4.1 Main. The **Main** routine conducts the simulation process and consists of the subordinate modules shown in Figure 4-2.

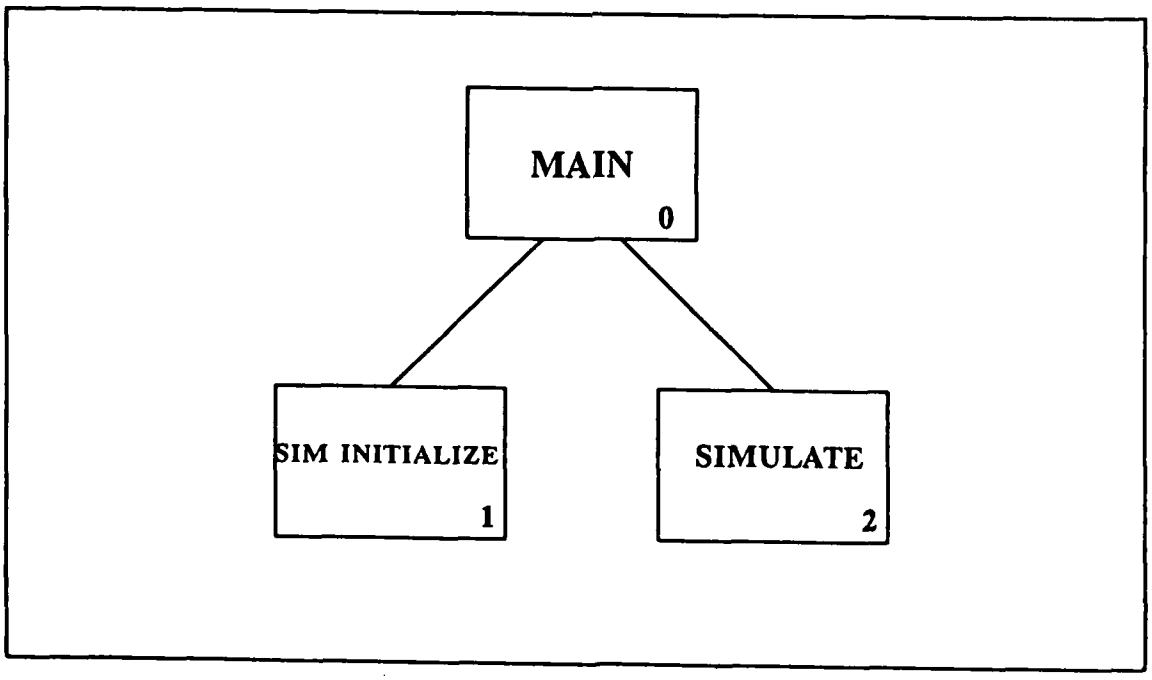


Figure 4-2 Main

The structured English description of the **Main** procedure is:

(0)* Main

1. Check command line validity.
2. Set option flags from command line options.
   - 2.1 They are:
     - -d n turn on debug option "n"
     - -s n start simulation at time "n"
     - -t n terminate simulation at time "n"
     - -o f output simulation results to file "f"
     - -b set breakpoint
       /* not implemented this version */
   - 2.2 None of the above
     - print error message
3. Get input filename from command line.
4. Create and initialize head and tail pointers to driver queue.
5. Set sort marker (**qmarker**) to driver queue head pointer.
6. Create and initialize simulation structures
   /* signals, drivers, transactions */
   /* behavior function pointers */
   /*See Figure 4.4 */
7. Execute simulation until done
   /* See Figure 4.12 */
8. At end print message

---

* Throughout the remainder of this chapter, the number(s) in parenthesis in the structured English description refer(s) to the related module in the accompanying hierarchical chart.

It should be noted that **driverhead** and **drivertail** are structures of type drive which are used as the head and tail of the event queue. For control purposes, the sort key field of **driverhead** is set to the VAX system's minimum integer value and the sort key field of **drivertail** is set to the system's maximum integer value.

Also note that **qmarker** is a pointer which marks the present position in the event queue. When the event queue is updated the sort is to the left (<=) or right (>) of qmarker.

4.4.2 **Sim Initialize**. **Sim Initialize** creates and initializes the simulation data structures, opens and reads the input vector file, and opens the output report. **Sim Initialize** consists of the subordinate modules shown in Figure 4-3.

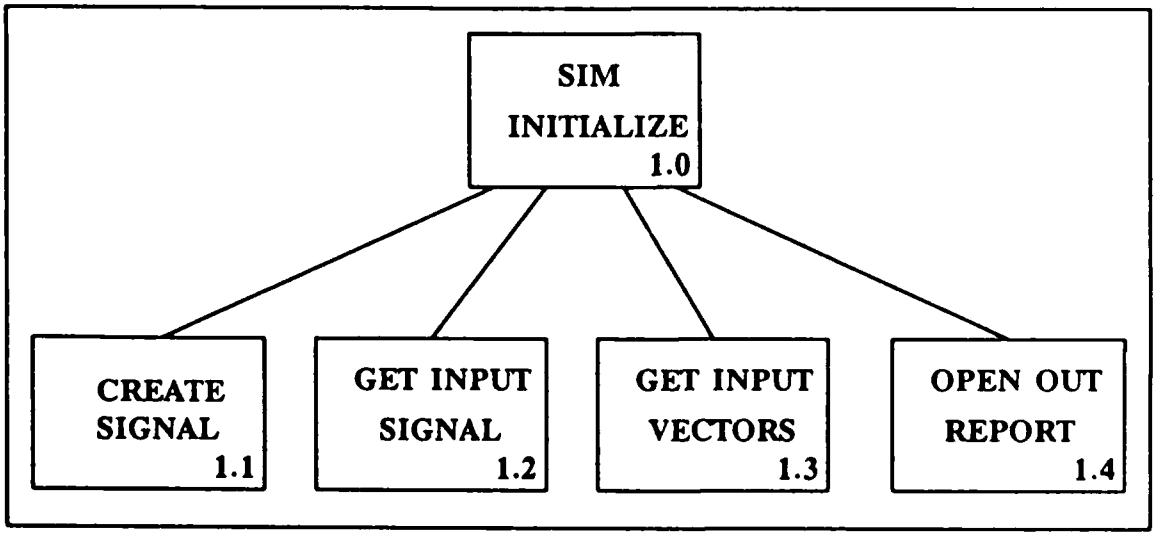


Figure 4-3 **Sim Initialize**

(1.0) Sim Initialize

1. Set simulation start time.
2. Set simulation termination time.
3. Create and initialize simulation structures.
   /* signals, drivers, transactions */
   /* behavior function pointers */
   /*see Figure 4.4 */
4. Read input signals.
5. Read input vectors.
6. Open the output file.
7. Return.

4.4.3 **Create Signal**. **Create Signal** opens and closes the data files and creates and initializes the signal, driver and fanout structures. **Create Signal** consists of the subordinate modules shown in Figure 4-4.

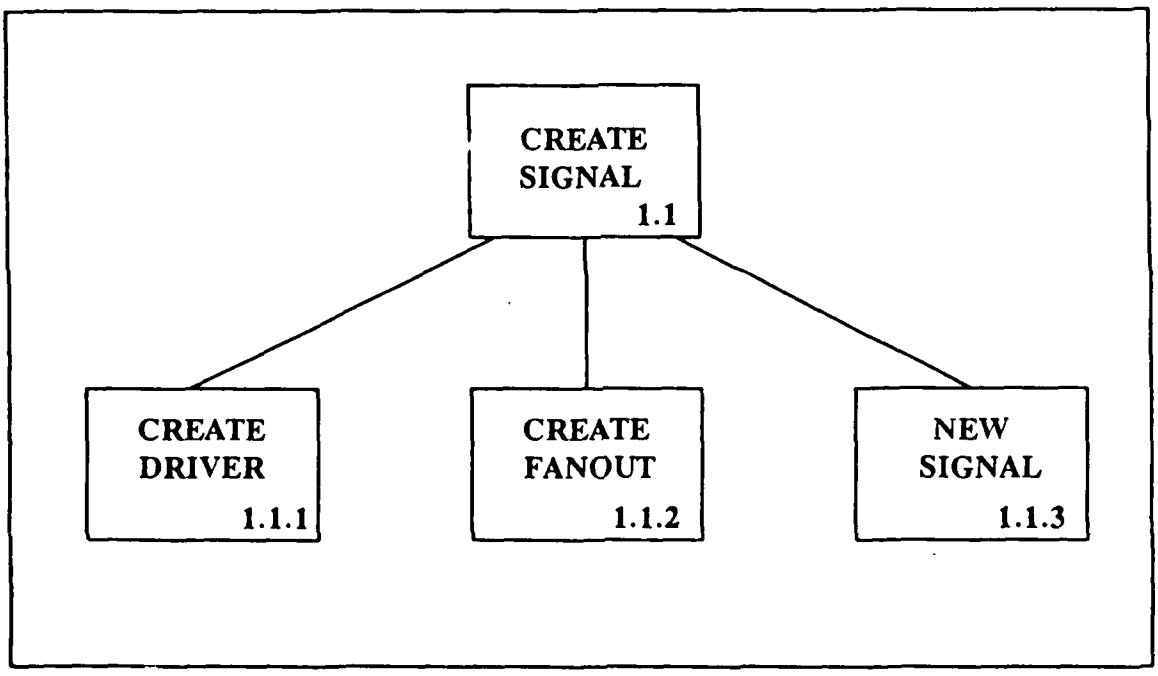


Figure 4-4 **Create Signal**

The structured English description of **Create Signal** is:

(1.1) Create Signal

1. Open the data files. These files contain the signal, driver and fanout data.

2. While there is more signal data do the following:
   2.1 Create signal structures
   2.2 Create driver structures
   2.3 Create fanout structures

3. Close the data files.

4. Return.

4.4.4 Get Input Signals. **Get Input Signals** opens the input vector file and reads the input signals from it, storing the signal address in an array. **Get Input Signals** consists of the subordinate module **Strcmp** shown in Figure 4-5.*

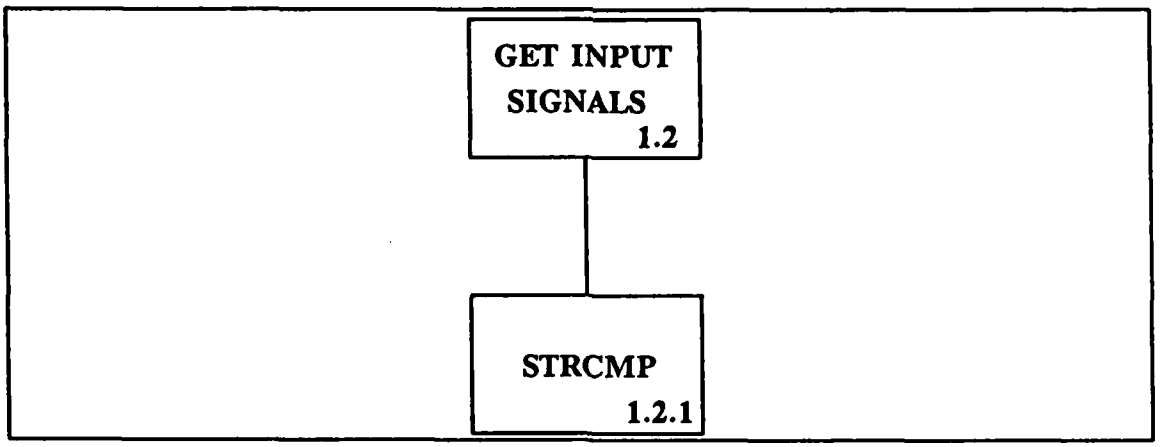


Figure 4-5 Get Input Signals

* The structured English description of the **strcmp** procedure shown in Figures 4-5 and 4-6 is not presented here due to its simplicity. A full description can be found in Kernighan and Ritchie (15:101).

(1.2) Get Input Signals

1. Open the input vector file.

2. If unable to open input file
   Print error message and terminate program.

3. While there are signal names to read do
   the following:

   3.1 Get the address of the signal and
   store it /* block 1.1.1 */

   3.2 If unable to get signal address
   print error message.

4. Return.

4.4.5 Get Input Vectors. Get Input Vectors reads the input vectors from the input file, performs a character to integer conversion and posts the input vector to the transaction queue of the parent driver. Get Input Vectors consists of the subordinate modules shown in Figure 4-6.

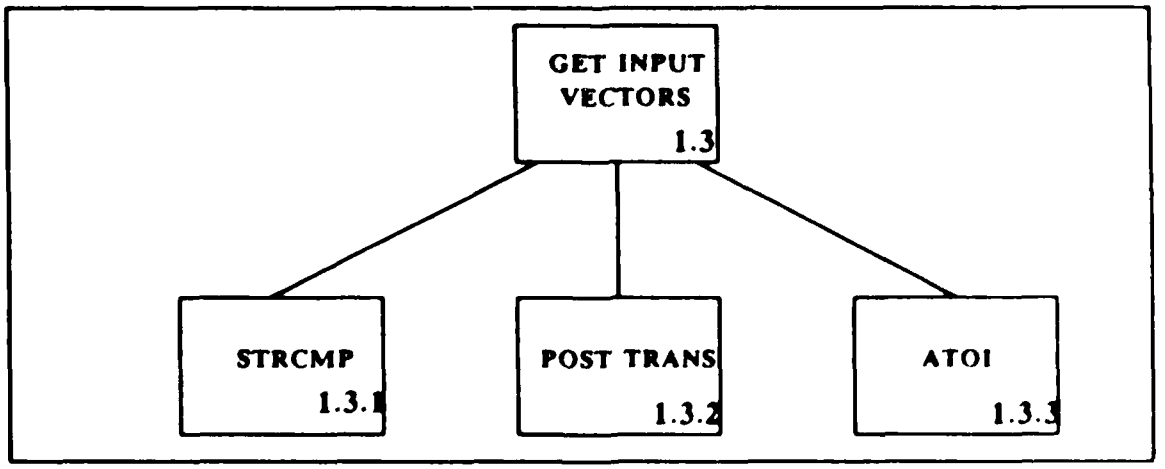


Figure 4-6 Get Input Vectors

The structured English description of **Get Input Vectors** is:*

(1.3) Get Input Vectors

1. Read the new input vector time.

2. While there are more vectors of the input vector time read do the following:

   2.1 Check the validity of the input vectors
   /* Block 1.2.1 */
   There are five:
   'u'
   'z'
   '0'
   '1'
   '.'
   /* '.' means no change to previous input vector value */

   2.2 None of the above:

   Print error message.

3. Get input signal address.

4. If the input vector value has changed create a transaction and post it.
   /* see Figure 4-18 */

5. Return.

4.5.6 **New Signal**. **New Signal** creates and initializes the signal structures. **New Signal** consists of the subordinate module **Strsave** shown in Figure 4-7.

---

* The structured English description of the **atoi** procedure shown in Figure 4-6 is not presented here due to its simplicity. A full description can be found in Kernighan and Ritchie (15:58). The structured English description of **Post Trans** in Figure 4-5 is presented in section 4.4.18.

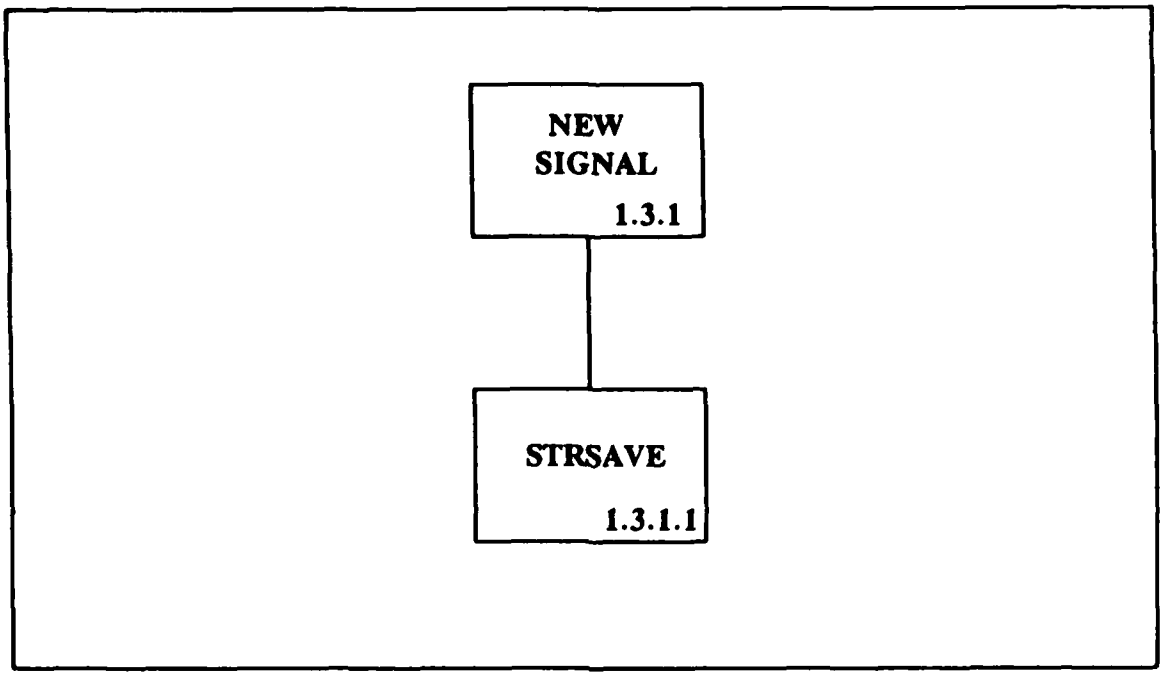


Figure 4-7 New Signal

The structured English description of New Signal is:*

(1.3.1) New Signal

Create and initialize new signal structure.

Return.

---

* The structured English description of the Strsave procedure shown in Figures 4-7 and 4-9 is not presented here due to its simplicity. A full description can be found in Kernighan and Ritchie (15:103).

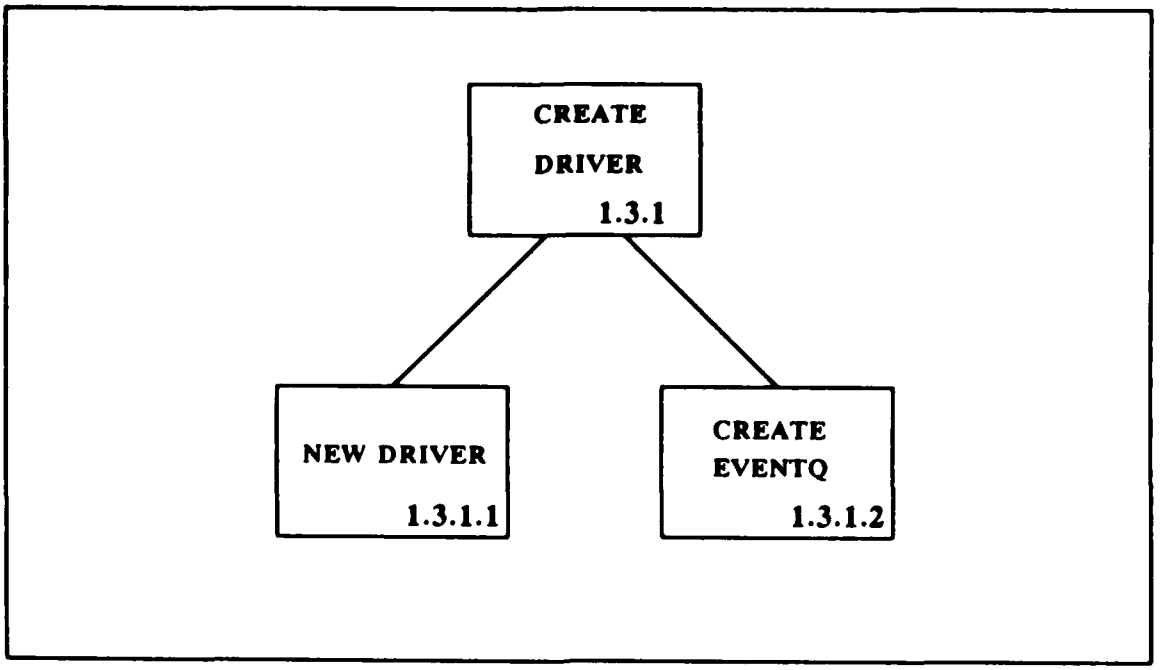


Figure 4-8 Create Driver

The structured English description for Create Driver is given below. The structured English description of Create event queue is given in section 4.4.15.

(1.3.1) Create Driver

1. While there is more driver data
   do the following:

   1.1 For each signal do the following:

      1.1.1 Read the driver name

      1.1.2 Create a new driver
            structure

      1.1.3 Create the event queue
            /* See section 4.4.15 */

   Return.

4.4.8 New Driver. New Driver creates and initializes the driver structures. New Driver consists of the subordinate module Strsave (15:103) shown in Figure 4-9.

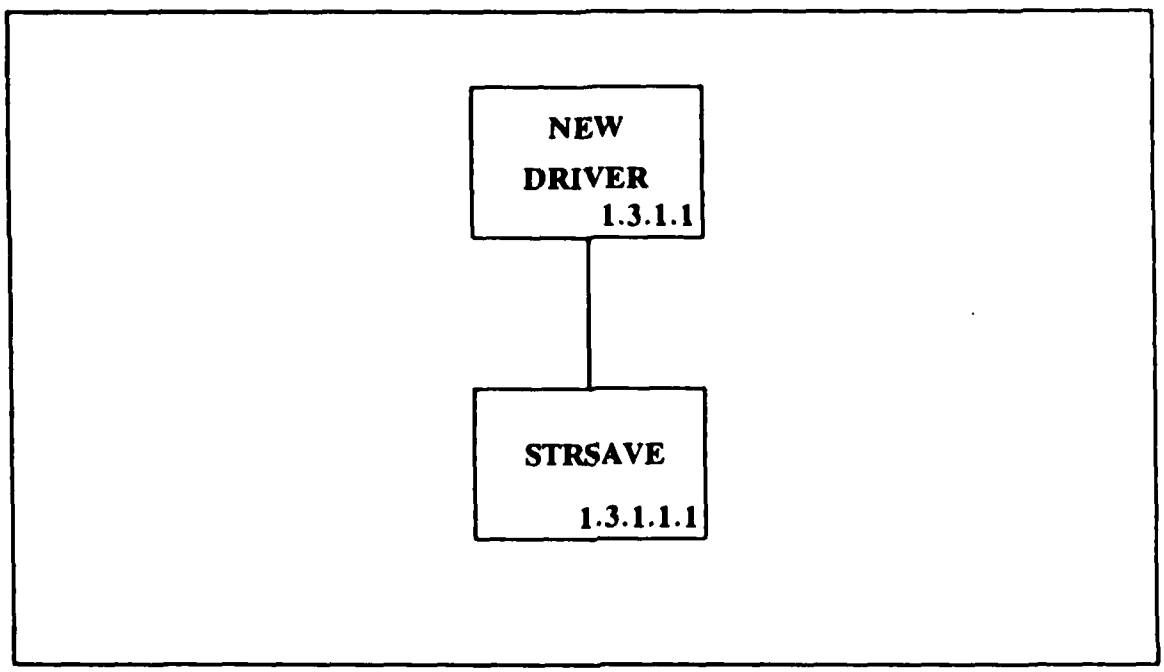


Figure 4-9 New Driver

The structured English description for New Driver is:

(1.3.1.1) New Driver

Create and initialize a
driver structure.

Return.

4.4.9 Create Fanout. Create Fanout reads the fanout data file and creates the fanout structure, storing the address of the behavioral function. Create Fanout consists of the subordinate module new fanout shown in Figure 4-10.

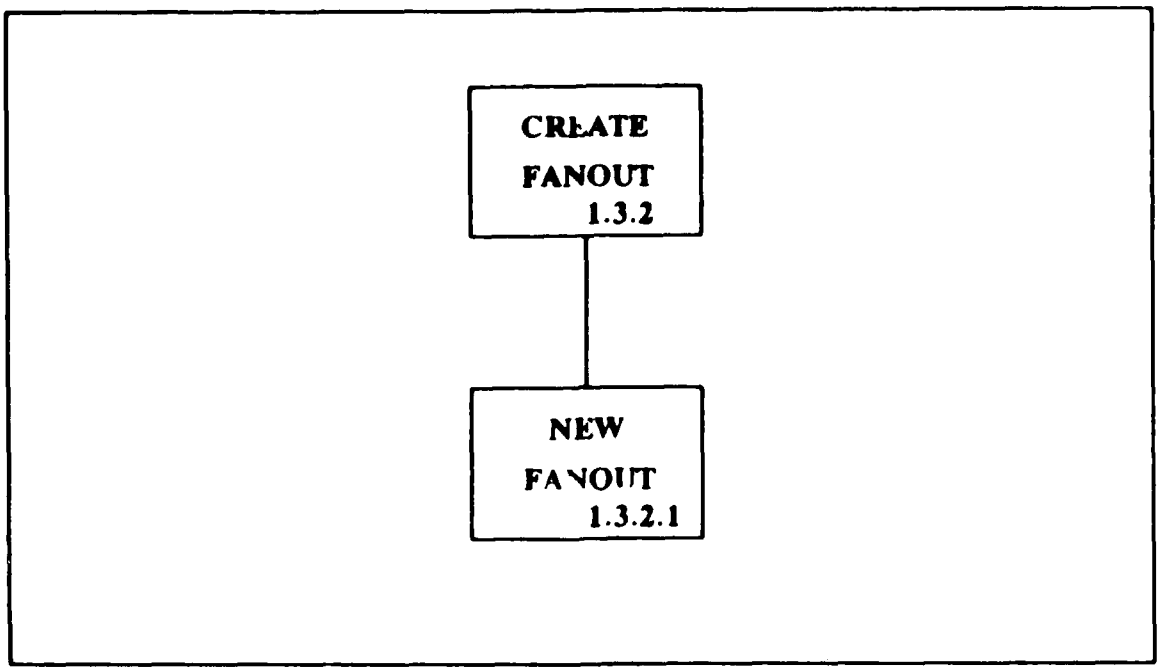


Figure 4-10 **Create Fanout**

The structured English description for **Create Fanout** is:

(1.3.2) Create Fanout

While there is more fanout data in the fanout file do the following:

For each signal with fanout do the following:

Get the behavioral function name.

Get the address of the behavioral function and store it.

Return.

4.4.10 **New Fanout**. New Fanout creates the fanout structure and stores the behavioral function address. New Fanout consists of the subordinate module get func shown in Figure 4-11.

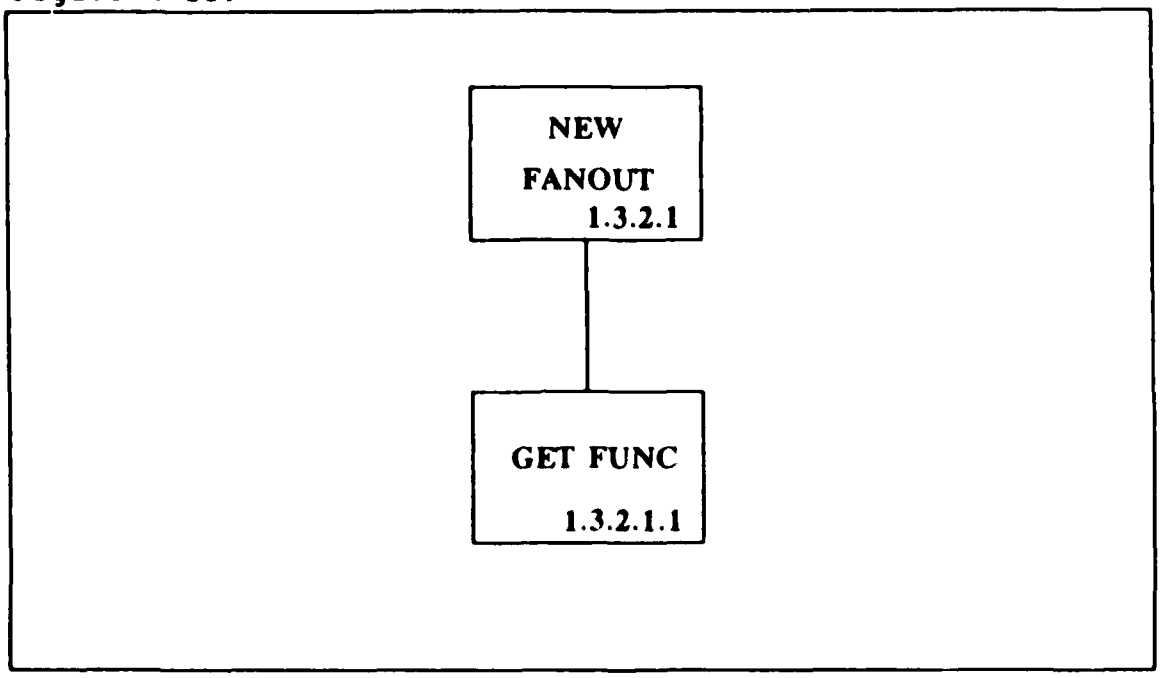


Figure 4-11 New Fanout

(1.3.2.1) New Fanout

Create new fanout structure

Get and store the address of the behavioral function.

Return.

Due to its simplicity, no further description of get func (block 1.3.2.1.1) is given. The interested reader should refer to the source code, which is published under separate cover as a technical report.

4.4.11 **Simulate**. Simulate schedules and executes the simulation. Simulate consists of the subordinate modules shown in Figure 4-12.

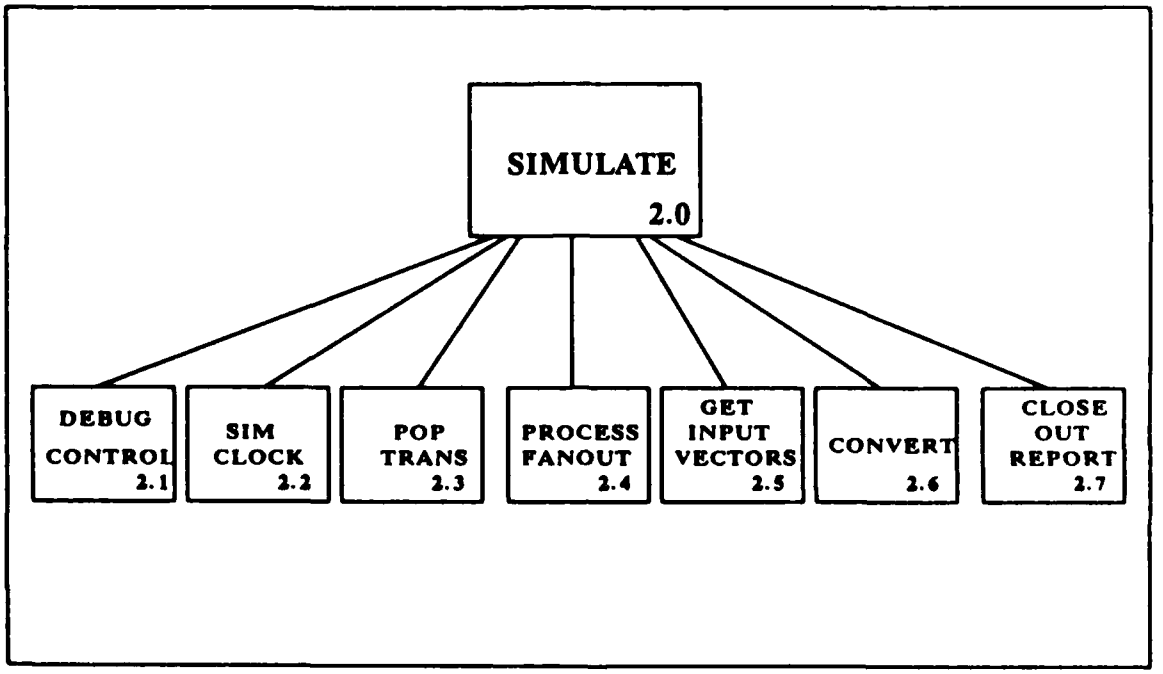


Figure 4-12 Simulate

The structured English description for Simulate is given below. All block references are to Figure 4-12.

(2.0) Simulate

1. Set first drive to point to the first event.

2. Set first trans to point to the first transaction of first drive.

3. If DEBUG is TRUE:

   3.1 Set the debug option.

   3.2 Print debug message.

4. While the simulation time is not finished and there are more events do the following:

   4.1 If the event time is less than the current simulation time:

      4.1.1 Print Warning Message

      4.1.2 Delete the transaction

   4.2 If the event time equals the current simulation time:

      4.2.1 If event

         4.2.1.1 Post new signal and driver value

         4.2.2.2 Print event data. /* time, old value, new, value, transaction type */

         4.2.1.3 Delete the transaction. /* block 2.3 */

         4.2.1.4 Propagate the new signal value to all outputs on the fanout list. /* block 2.4 */

      4.2.2 If no event:

         Delete the transaction.

4.3 If the event time does not equal current time:

Increment the simulation clock. /* block 2.2 */

4.4 If the transaction processed was an input vector:

read next input vector.

4.5 Update **first drive** and **first trans**.

5. If Done:

5.1 Print report summary.

5.2 Return.

Due to the simplicity of the **sim clock** (block 2.2) and **convert** procedures (block 2.6), no further description is given. The interested reader should refer to the source code.

4.4.12 **Debug Control**. **Debug Control** is only used for checking the correctness of all or part of the program during development or after program modification. VSIM does not require **Debug Control** for execution. **Debug Control** consists of the subordinate modules shown in Figure 4-13.

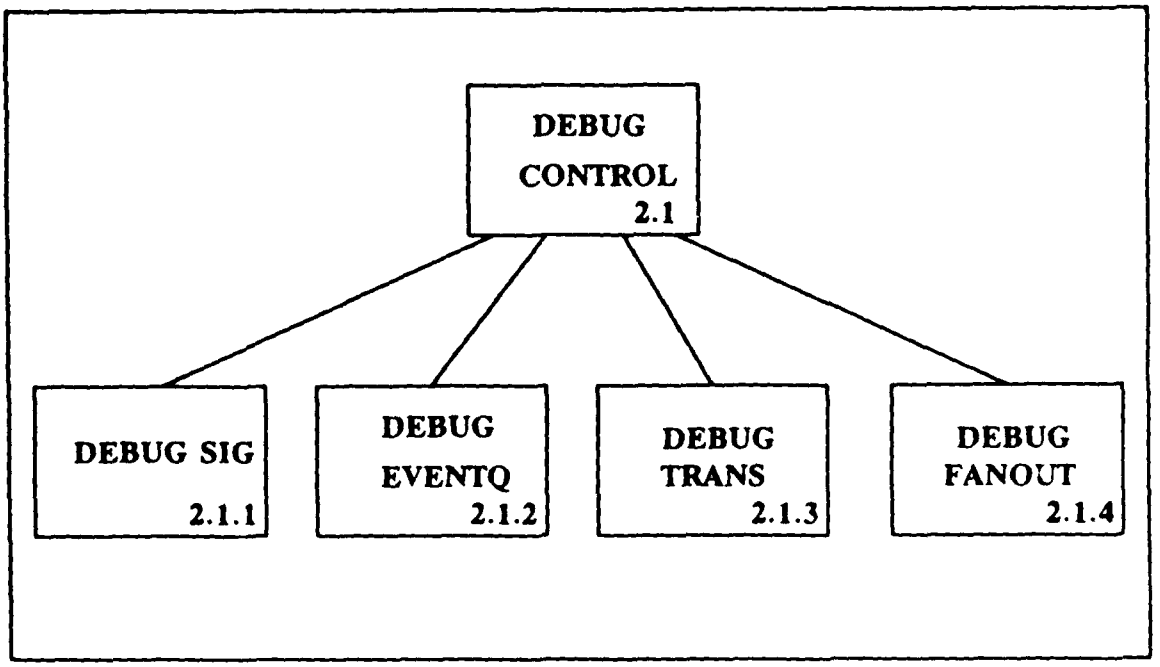


Figure 4-13 **Debug Control**

The structured English description for **Debug Control** is:

(2.1) Debug Control

```
1.  Set the debug option from the
    command line.

    There are four cases:

    Signals (1) dump the signal structures.

    Eventq  (2) dump the eventqueue.

    Trans   (3) dump the transaction queues.

    All     (4) dump all the queues.
    /* signal, event, transaction, fanout */

    None of the above

        Print error message

Return.
```

Since they are not required for program execution, no further description of the debug functions (blocks 2.1.1 - 2.1.4) is given. The interested reader should refer to the source code.

4.4.13 **Pop Trans**. **Pop Trans** deletes a transaction and frees the memory space. **Pop Trans** consists of the subordinate module **Updateq** shown in Figure 4-14.

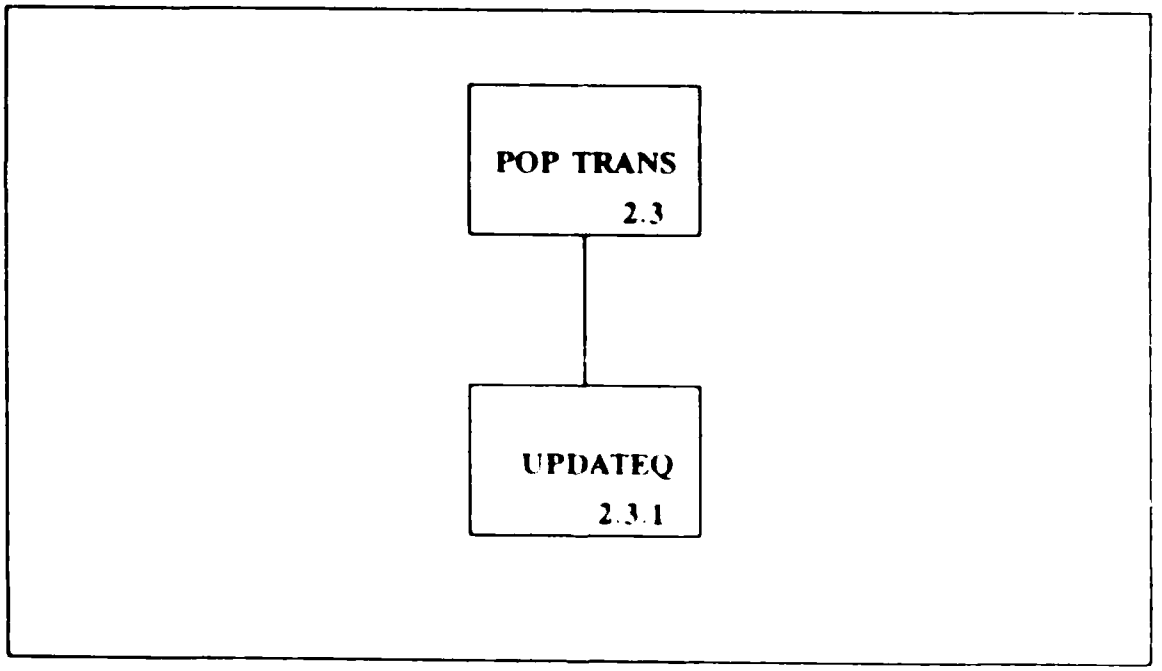


Figure 4-14 **Pop Trans**

The structured English description for **Pop Trans** is:

(2.3) Pop Trans

1. Set **first drive** to point to the first event.

2. Set **first trans** to point to the first transaction of first drive.

3. If **first trans** equals the **null trans**

   Print a Warning Message.

4. Remove the transaction from the queue.

5. Resort the event queue.
   /* block 2.3.1 */

6. Free the memory space.

7. Return.

4.4.14 **Updateq**. **Updateq** resorts the eventq. **Updateq** consists of the subordinate module **Create eventq** shown in Figure 4-15.

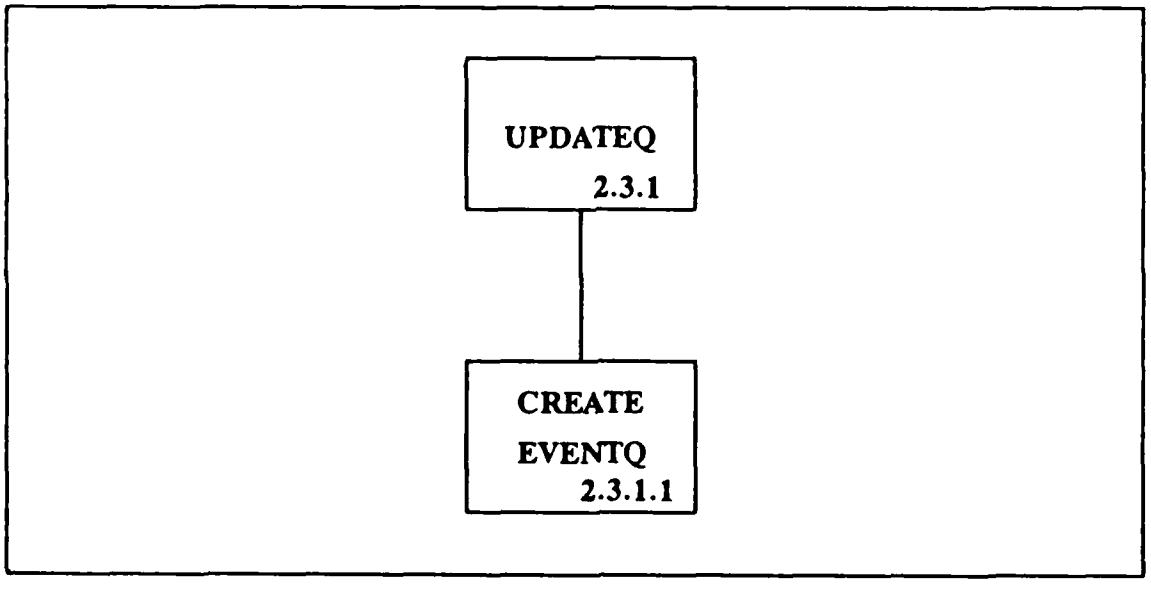


Figure 4-15 **Updateq**

(2.3.1) Updateq

1. Remove the event from the event queue.

2. Put event back in event queue based on its sort key.
   /* **Create Eventq.** See section 4.4.15 */

3. Return.

4.4.15 **Create Eventq**. The structured English description for **Create Eventq** is:

(2.3.1) Create Eventq

1. Set the sort key of event to the new sort key value.

2. If the sort key of event is less than or equal to the **qmarker** sort key do the following:

   2.1 Search the event queue in the direction of **drivertail** until the sort key of event is greater than the **qmarker** sort key.

   2.2 Put the event back in the queue.

   2.3 Set **qmarker** to point to the event.
   /* sets a new sort marker for the queue */

   2.4 Return.

3. Else

   3.1 Search the event queue in the direction of **driverhead** until the sort key of event is less than or equal to the **qmarker** sort key

   3.2 Set **qmarker** to point to the event

   3.3 Return.

The search mechanism used is intended to decrease the sort time of a sequential search. Although not a true binary search, it does decrease the search time.

4.4.16 Process Fanout. Process Fanout propagates the value of an event to all the outputs on the fanout list. Process Fanout consists of the behavioral function modules shown in Figure 4-16.

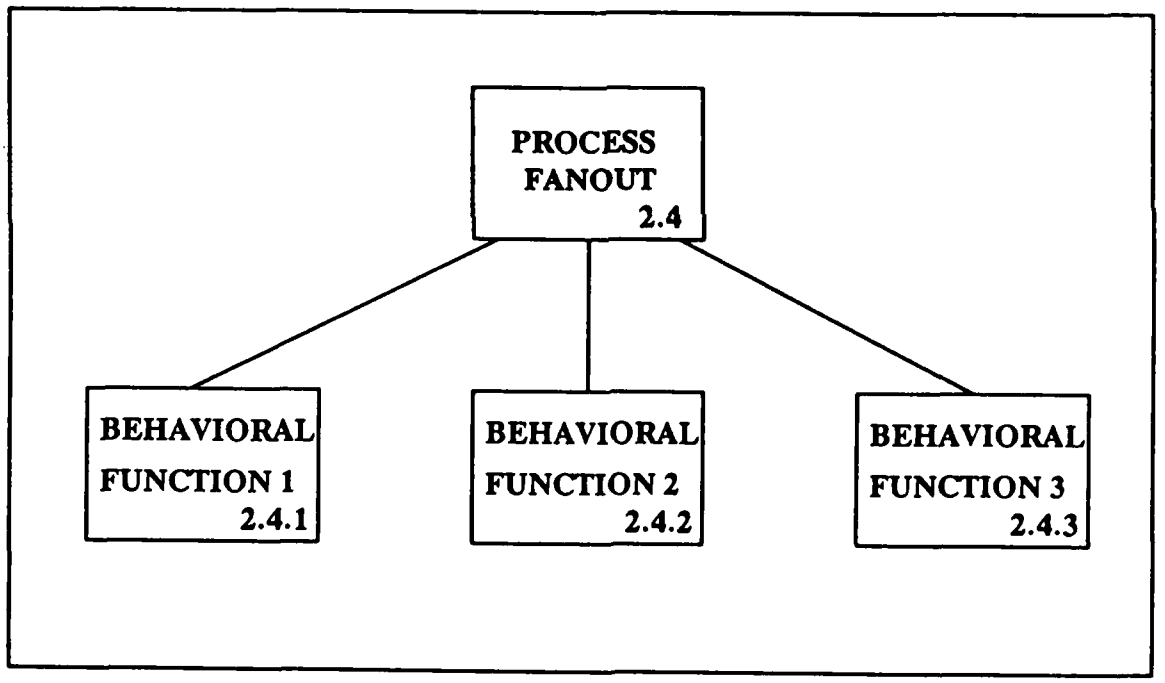


Figure 4-16 Process Fanout

The structured English description for Process Fanout is:

(2.4) Process Fanout

1. While there are more outputs on the signal fanout to propagate do:

   Execute the behavioral function

2. Return.

4.4.17 Behavioral Functions. The behavioral functions calculate the future values and times for target signals on the event fanout list. The behavioral function modules consist of the subordinate modules shown in Figure 4-17.

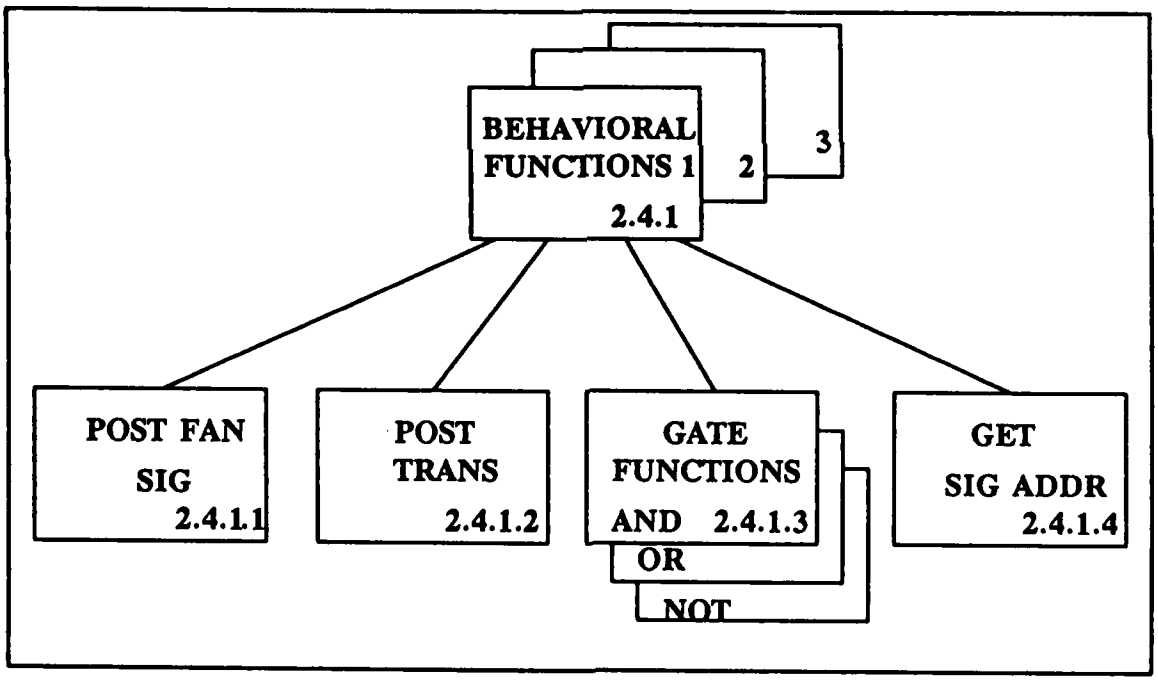


Figure 4-17 Behavioral Functions

Since the behavioral functions are the same except for variable values and execution of different generic gate functions, only one representative structured English description of a behavioral function is presented.

(2.4.0) Behavioral Functions

1. Get current value of drivers.
2. Calculate the future value and time for the target output signal.
   /* the generic functions and, */
   /*or, not calculate the future value*/
   /*of the specified gate type*/
3. Post the future time and value to the appropriate driver of the target output signal.
4. Return.

Due to their simplicity, no further description of post fan sig (block 2.4.1.1) and the generic gate functions (block 2.4.1.3) is given. The interested reader should refer to the source code. Post Trans is described in section 4.4.18.

4.4.18 Post Trans. Post Trans creates a new transaction structure, inserts it in the appropirate transaction queue and deletes old transactions depending on the type delay as required. Post Trans consists of the subordinate module Updateq shown in Figure 4-18.

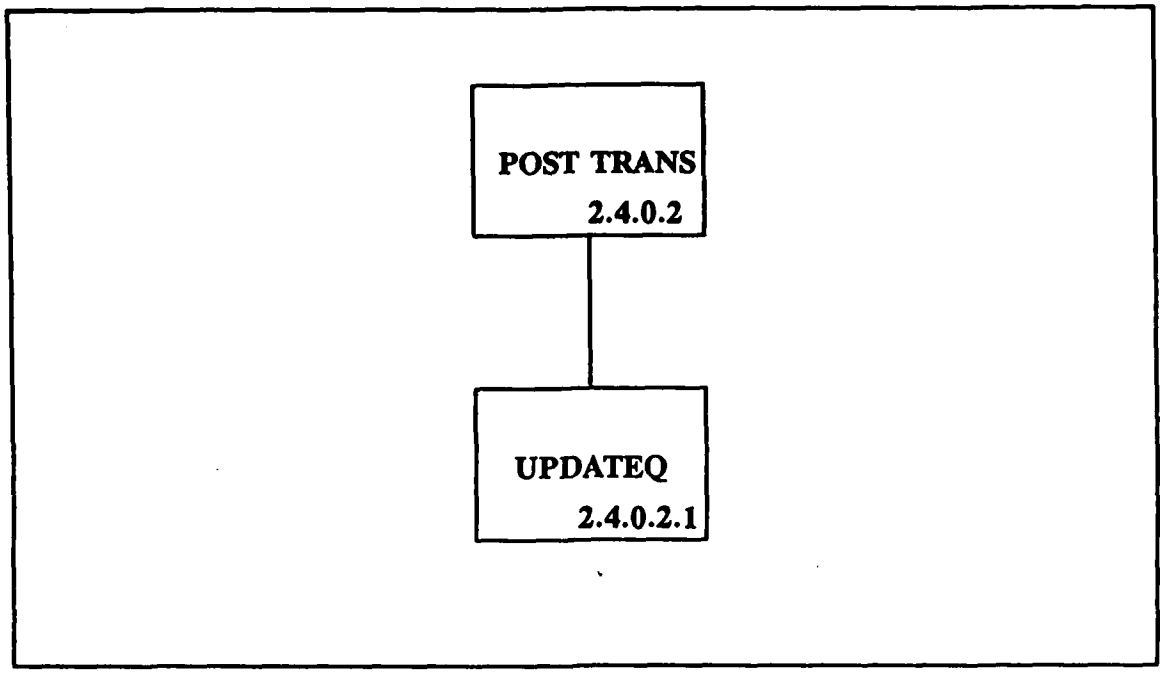


Figure 4-18 **Post Trans**

The structured English description for **Post Trans** is:

(2.4.0.2) Post Trans

1. Create and initialize a transaction.

2. Insert the new transaction in the transaction queue.

3. If the new transaction is not an input vector do the following:

   Update the projected output waveform
   /* See note below */

4. If the new transaction is the first transaction in the queue:

   Resort the event queue
   /* See Figure 4-15 */

Updating the projected output waveform consists of deleting zero or more previously computed transactions depending on the type of delay. VHDL supports transport and initial delay and VSIM fully implements this function. A detailed explanation can be found in (13:85).

V. Analysis and Results

5.1 General

The results of this thesis effort and an analysis of those results is the objective of this chapter. This provides the author the opportunity to not only critique his research effort but to highlight the strengths and weaknesses of the design and the program itself.

The method used to perform this analysis was to analyze several areas of the program's design and function, and then present and evaluate the results produced by the program.

The following specific program areas were analyzed:

1. The quality of the program design was compared with the initial design goals.
2. The program's function was compared to the functional requirements.
3. Program performance was compared to the performance requirements.

The reader is reminded that this thesis was concerned with the **Sim Initialize** module and the **Simulate** module (Blocks 2 and 3 of Figure 3-2). A detailed evaluation of VSIM for different circuit structures, gate delays and fan-in and fan-out was not done due to the desire to evaluate the program's function in the limited thesis time

available. Two different circuit structures were evaluated, however, as were the program's execution speed versus the number of input test vectors and the modeling of different gate delays.

5.2 VSIM Design

The program design is specific and detailed concerning all aspects of VSIM. The design hierarchy is well-documented in a complete set of hierarchical charts (Chapter 4) and the C code itself includes detailed headers which provide important information and a description of each program module.

The code is understandable and well-designed. However, due to the time constraints under which the program was designed and developed, it is quite possible that some of the program modules could be revised to gain increased efficiency in execution speed and use of memory space. An example is the event queue which is implemented as a doubly-linked list and is designed to function as a modified binary search. This search algorithm could be replaced by a more efficient one.

Names for the functions and variables were selected to be appropriate and descriptive of the required function and data represented; however, there are instances where the function or variable name is misleading or was poorly

chosen. Functions which are poorly named are generally ones whose purpose was modified during program development. An example is the **Create Signals** module which would be more appropriately named **Create Structures**.

Since the program was designed to minimize the size of the executable file, several functions are used more than once rather than duplicating a similar function among several slightly different program modules. Examples of this modularity and efficiency are the **Post Trans** and **Create Eventq** functions.

5.3 VSIM Function

VSIM implements each of the functional requirements specified in Chapter II. The following functions are performed by VSIM:

5.3.1 Operation. VSIM reads and interprets the command line, the data files and the input vector file to create the program data structures and establish the runtime environment. The hardware design is then simulated until all transactions are processed or a preset termination time is reached; events are evaluated, new transactions created, and results reported in the output trace file that is generated.

5.3.2 VHDL Implementation. VSIM implements a subset of VHDL. VSIM supports single drivers, simple signal assignment statements, and single block statements. VSIM supports integer and floating point data types, although the only input values coded in the behavioral functions are: '0,' '1,' 'z' (high impedance), and 'u' (unknown). The VHDL functions of inertial and transport delay are fully implemented.

5.3.3 Interactive Capability. The user designates the input vector file. VSIM also accepts and implements the following command line options:

-d selects debug.

-o designates the output trace file.

-s selects the simulation start time.

-t sets the simulation termination time.

These functions have been verified through extensive testing. Each option was used alone and in conjunction with other options.

5.3.4 Error Checking. Error messages are generated for syntax and semantic errors. Error messages are also generated for errors in the command line. All error messages were exercised and checked.

5.3.5 Program Size. The VSIM program consists of 1250 lines of C source code. The compiled object code is 24K bytes in length.

5.4 Program Results

VSIM was exercised many times to verify program correctness and to detect any program "bugs." Although considerable testing was accomplished considering the time available and the requirement to manually code the design data needed to drive the simulator, additional testing could be performed to further insure program correctness. The tests conducted on VSIM and the results achieved are presented and analyzed in the following sections.

5.4.1 Designs Simulated. Figure 5-1 shows the two circuits that were used to drive VSIM. These circuits were used primarily to check the VSIM program operation. The delay for each gate is shown above the gate and if transport delay a T label is indicated below the gate. In the absence of transport delay, all gates have inertial delay (the default case).

The three-gate circuit (Figure 5-1a) implements the basic gate functions **And**, **Or**, and **Not**, while the six-gate circuit (Figure 5-1b) models a simple combination of two copies of the three-gate circuit. Both circuits were driven

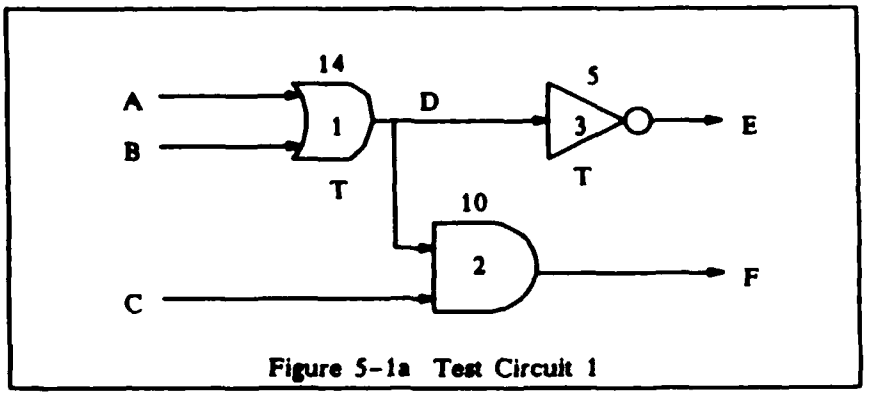


Figure 5-1a Test Circuit 1

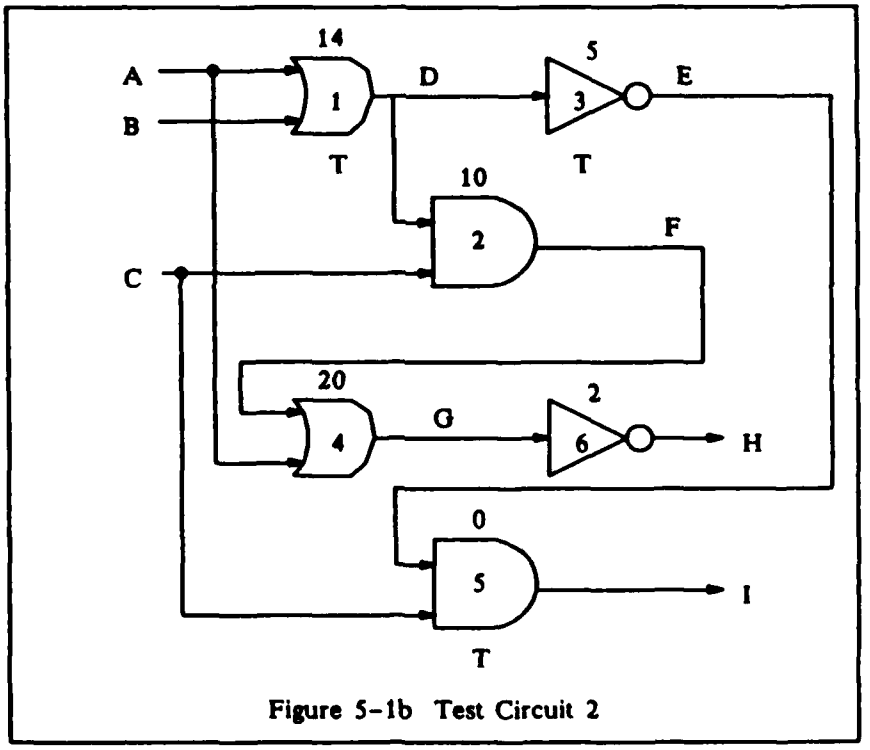


Figure 5-1b Test Circuit 2

Figure 5-1 Test Circuits

by identical input vector files to provide a common base for analyzing simulation results. The input vector files contained the five permissible input test vector characters ('0,' '1,' 'z,' 'u,' '.') where '.' means use previous vector value. The input test vectors were randomly selected. The number of input test vectors was varied from a minimum input of 10 to a maximum of 960. Appendix C contains an example of one of the input vector files used. The different delay types and times were used to check VSIM's capability to handle the full set of delay types and times.

5.4.2 Program Correctness. The output trace file produced by VSIM, when driven by the different input test files for the two circuits simulated, was carefully analyzed to validate that the program was producing the desired results. The output produced by VSIM for each circuit design and known input was checked using the boolean expression for the circuit simulated. In each test case for the different input vector test files and the two circuit designs under simulation, VSIM produced the correct results. Appendix H contains the representative results of one of the validation test runs done for the 3-gate circuit design.

The results also validated that VSIM correctly models both inertial and transport delay and correctly updates the

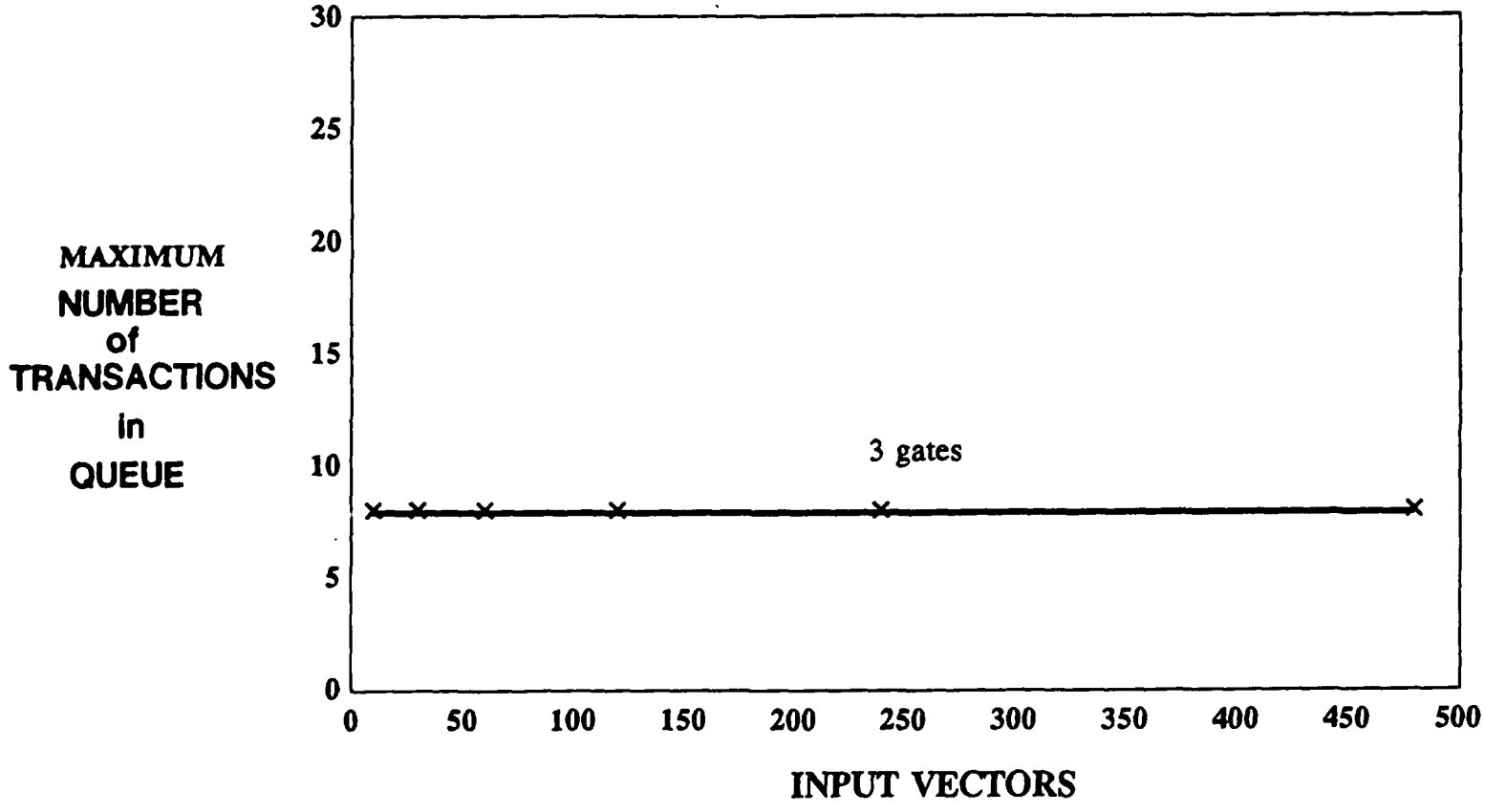
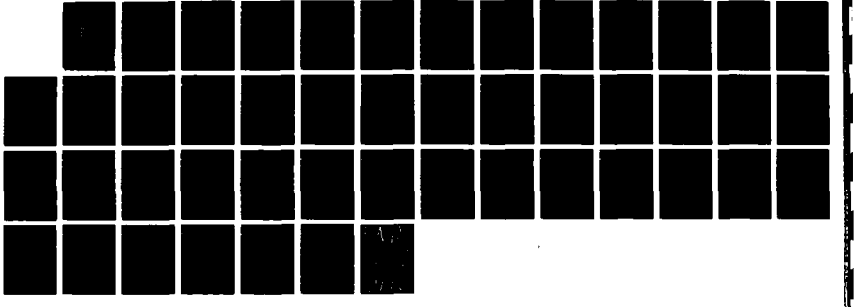
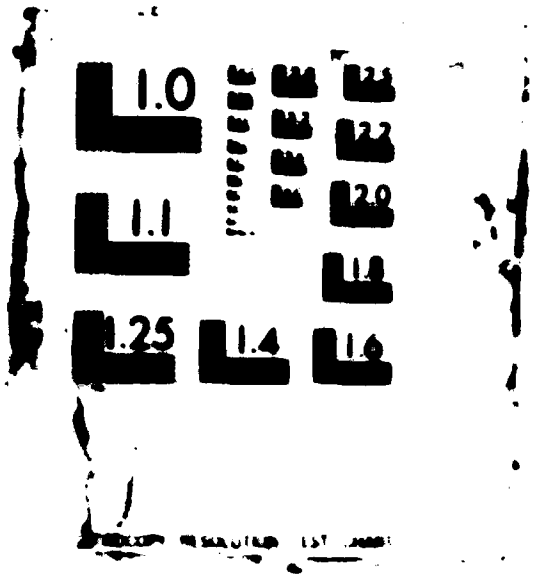


Figure 5 - 2 Depth of Transaction Queue

AD-A179 073 VHDL (VHSIC (VERY HIGH SPEED INTEGRATED CIRCUITS) 2/2
HARDWARE DESCRIPTIVE LA..(U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. W L LYNCH
UNCLASSIFIED DEC 86 AFIT/GCS/ENG/86D-15 F/G 9/2 NL




1.0
1.1
1.25
1.4
1.6
2.0
1.8
2.2

output wave form when posting new transactions. As shown in Figure 5-1b, a delay time of zero was simulated on gate 5, to verify that VSIM was correctly processing new transactions scheduled for delta time.

5.4.3 Transaction Queue Size. Figure 5-2 shows the maximum transaction queue size for six different input vector test files simulated on the 3-gate circuit design and is representative for both test circuits simulated. The significance of the data presented in Figure 5-2 is that it:

1. Demonstrates the efficiency of the VSIM design in processing the input vector test files. VSIM conserves memory space by reading in the input test vectors one line at a time as they are required by program execution. Reading in all the input test vector file would consume vast amounts of memory and create an inefficient data structure. Besides the efficiency question of not reading the entire input vector test file is the question of practicality when processing input vector test files for VLSI class designs which could contain tens of thousands of input test vectors.

2) Demonstrates that VSIM's dynamic data structures can support the changing data requirements of simulated designs during simulation execution.

3) Shows data structures are efficient in the use of scarce memory space - expanding to handle the required data and contracting to conserve space when the data requirements lessen.

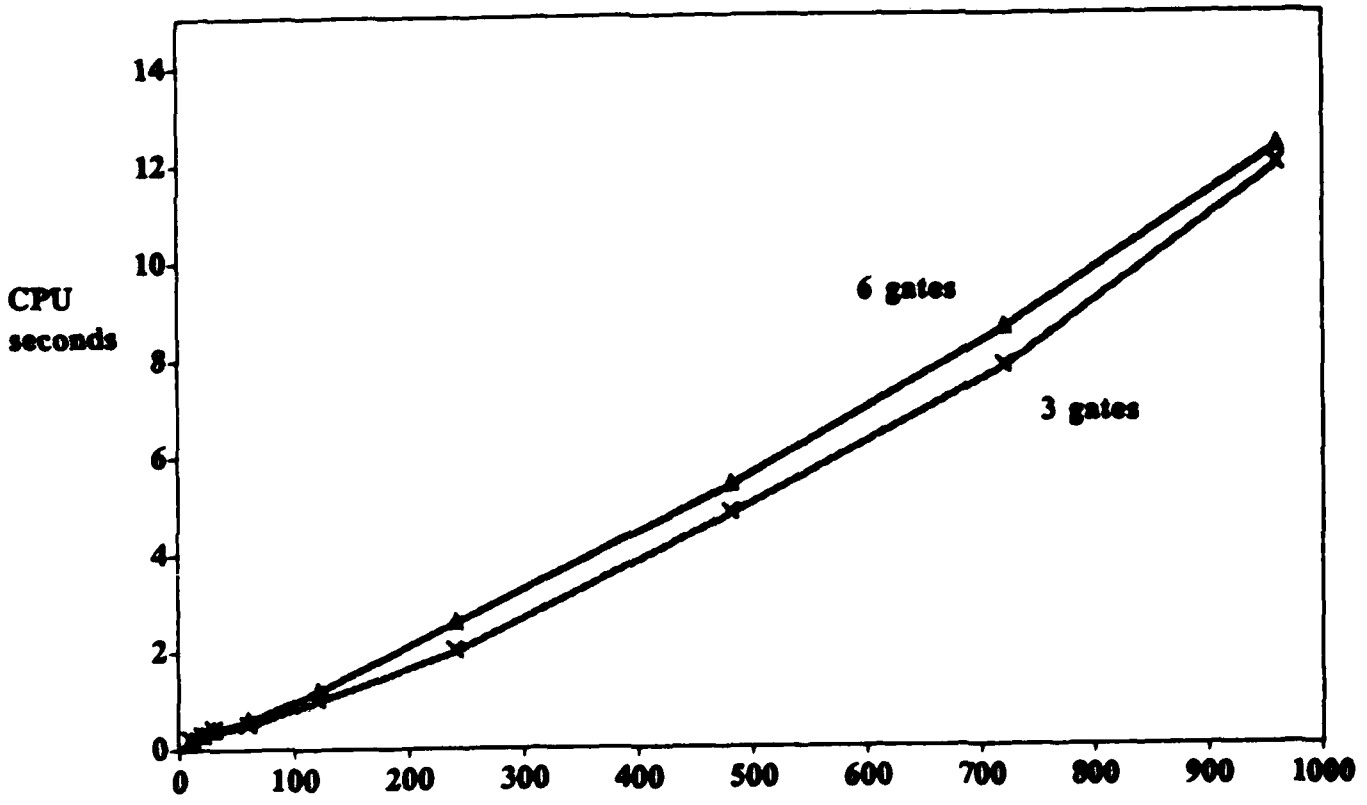


Figure 5 - 3 Simulation CPU Time

5.4.4 Simulation CPU Time. Figure 5-3 presents the simulation execution time for the two designs simulated.* As demonstrated by this data, VSIM appears to be functioning efficiently for the algorithm used and the two designs simulated. Based on this data the simulation execution time seems to be increasing in a linear manner as expected. This efficiency in execution time is required when simulating VLSI class chips, and demonstrates that VSIM is functioning consistent with its intended design.

Based on the data presented in Figure 5-3, it can be concluded that VSIM functions with generally the same efficiency regardless of the size of the input vector file or the number of gates in the design simulated. It must be cautioned, however, that these conclusions are based upon the limited number of tests conducted on VSIM. More testing is needed to verify these test results and validate VSIM. If additional testing verifies these results, then the data obtained in Figure 5-3 can be used by the designer to estimate the VSIM execution time of a design with a known number of input test vectors. An even more important result, however, if these data hold is that the fully implemented VHDL simulator using the same algorithm as

---

* During the testing a variance was discovered in the CPU time recorded by the system; therefore, the CPU times should be considered approximate and not exact.

VSIM should have the same efficiency. This is important since it provides the designer of the fully implemented simulator with a baseline from which to work. Implementation by the designer of a different algorithm to improve the efficiency of either VSIM or the fully implemented simulator can be measured in terms of this established baseline.

5.4.5 Events Processed. Figure 5-4 shows the number of events processed per CPU time measured against the number of input test vectors. For both circuits the number of events/second expands to a maximum rate and then gradually declines to a steady state range for the remainder of the simulation runtime. It is suspected that this rapid increase in events processed per second at the low end of the range of input vectors is caused by 1) the uncertainty of the system time routine to accurately calculate small time values, and 2) the time required to initialize the simulation as opposed to the actual execution of the simulation is the predominate contributor to execution time at the low end of the range.

The number events processed per second is not only directly related to the efficiency of the simulator, but is also highly dependent upon the design simulated and input vector test file used. As stated in section 5.4.4 above, the number of events processed per second can also be used as a measure of the efficiency of the simulator.

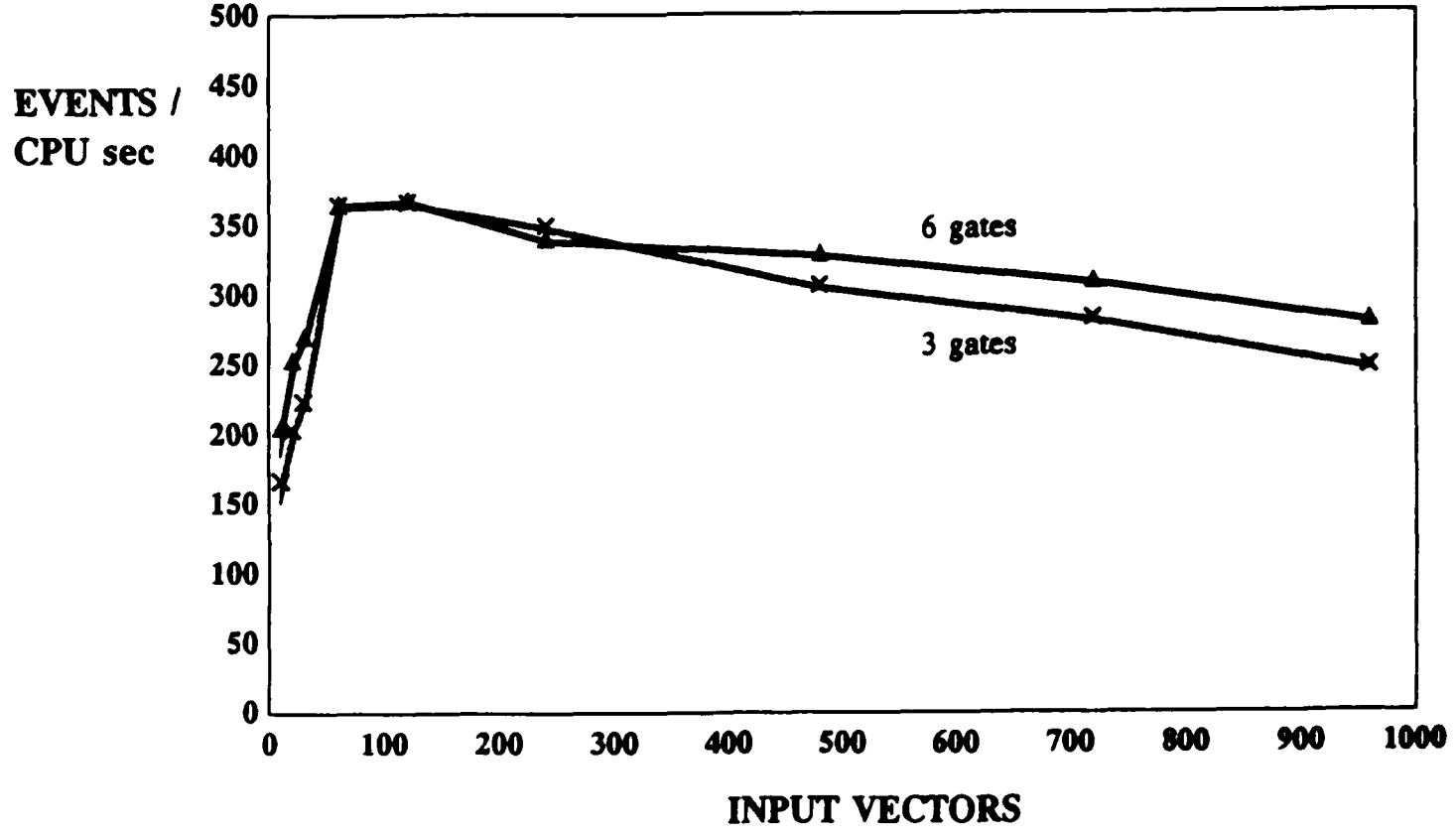


Figure 5 - 4 Events

5.4.6 Transactions. Figures 5-5 and 5-6 show the total number of transactions processed and the number of new transactions created per second during the simulation test runs. The explanation provided in section 5.4.5 concerning the rapid increase in events processed per second at the low end of the range is also applicable for the two cases discussed here. The significance of the number of transactions created can be compared to the number of events processed to obtain an indication of how many created transactions actually become events. This data is presented not only to provide other measures of the efficiency of the VSIM but to demonstrate that VSIM provides useful data in its output trace file.* Using this information in combination with the data presented in sections 5.3 and 5.4 above, the designer can adapt his approach to modeling to improve the simulation runtime and make the simulator more efficient.

* Appendix G contains an example of an output trace file generated by VSIM.

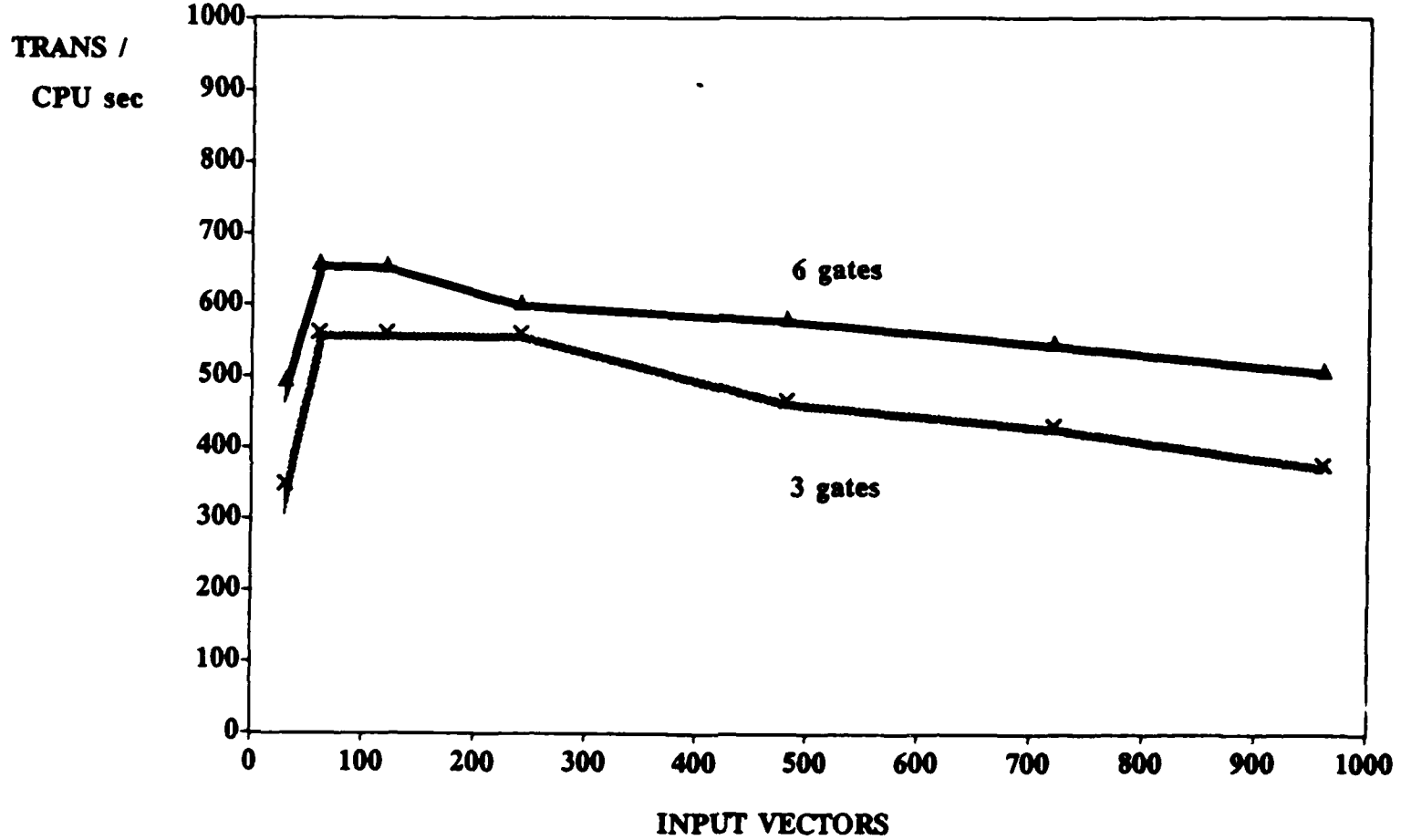


Figure 5 - 5 Total Transactions Processed

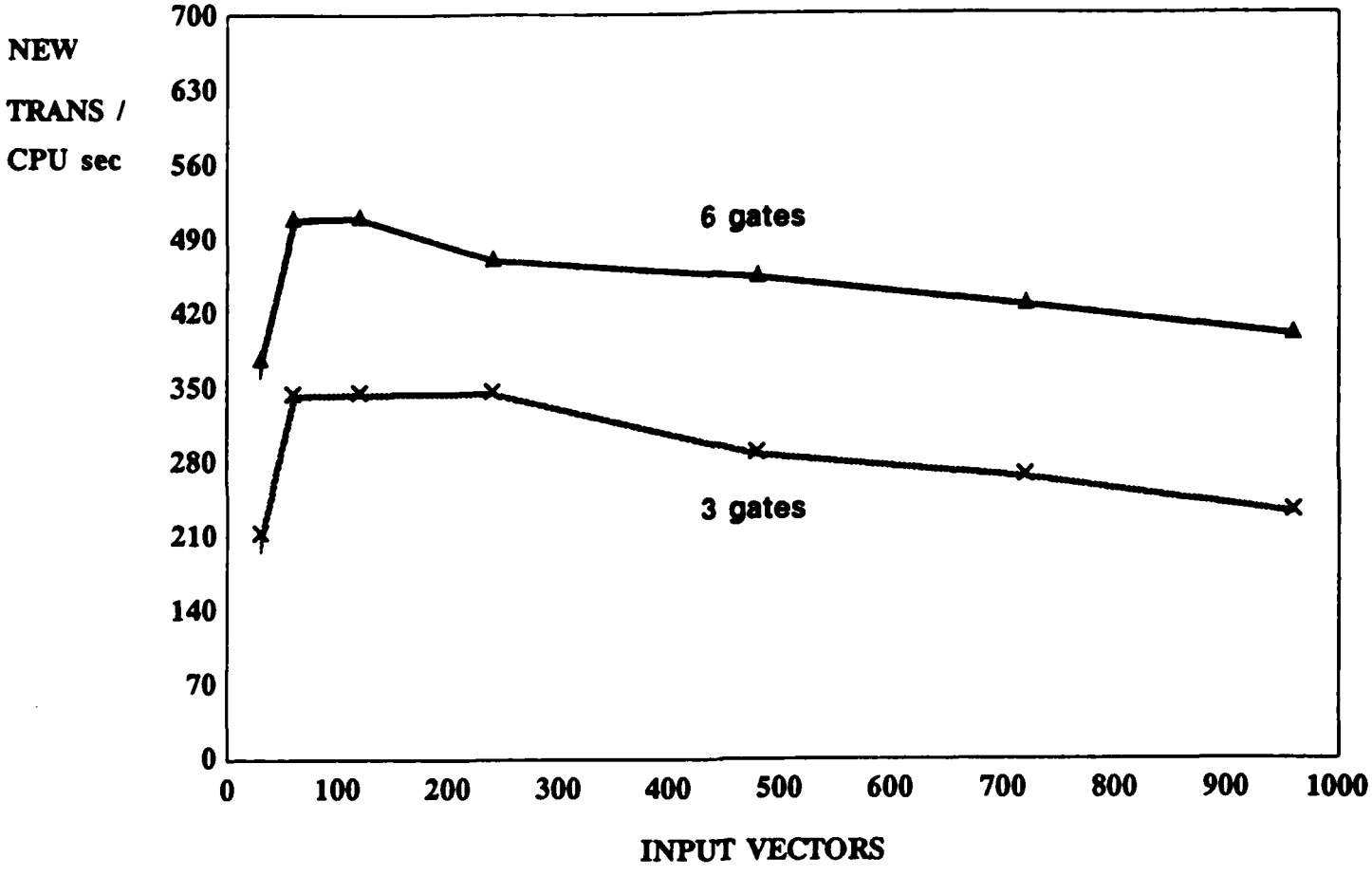


Figure 5 – 6 Transactions Created

5.5 Summary

The results and analysis presented demonstrate that VSIM meets the goals and functional requirements specified. An analysis of the program results, obtained from the various tests conducted and described within this chapter, provides proof of program correctness and validates the VSIM design concept.

VI. Conclusions and Recommendations

6.1 General

The purpose of this thesis was the development of a prototype VHDL simulator in support of the AFIT VHDL Environment (AVE). The prototype simulator implemented a simple signal structure and manually coded behavioral functions representative of VHDL processes. The prototype kernel simulator which was developed illustrates the basic simulation capabilities required for VHDL. The prototype simulator is the first step in the development of a complete simulator for the AVE. The added enhancements needed to upgrade the prototype are presented in Section 6.3, Recommendations. Evaluation of the prototype simulator kernel demonstrates that the anticipated runtime for the fully implemented simulator, to be designed and implemented on the UNIX system, should have excellent performance characteristics.

6.2 Conclusions

The VSIM program successfully implements a prototype VHDL simulator and provides excellent proof of design concept for implementing a complete simulator in the UNIX

environment. In general, VSIM meets all the established functional requirements for the prototype simulator and meets or exceeds initial performance expectations. The final VSIM program is highly modularized and is efficient both in memory usage and execution speed. The executable C file (object code) for VSIM is a compact and efficient 24K. VSIM executed the test circuit designs and input test vector files well within acceptable execution times. The output trace file produced by VSIM provides the designer with useful and required information which can be used to improve simulation modeling and efficiency.

Most importantly, the prototype simulator, VSIM, has provided: 1) a proof of design concept for development of a complete VHDL simulator for the UNIX Environment, and 2) an established baseline upon which future research and development efforts can build.

6.3 Recommendations

The primary recommendation of this thesis is that the research and development of a complete simulator for the AFIT VHDL Environment continue. The following recommendations focus on what remains to be done in the development of the simulator.

6.3.1 Parallel Development. VIA is now developed sufficiently that it is capable of driving the prototype simulator kernel. This is a significant milestone in the AVE effort and the development of VIA and the simulator should be one of parallel development. As VIA is expanded to incorporate new capabilities, the simulator should be concurrently expanded to incorporate the added features.

6.3.2 Translator. The translator (preprocessor) or automatic behavioral function generator for converting VIA structures into C functions which are integral to the simulator needs to be developed. A brief discussion of the translator was provided in Section 3.4, Chapter 3, System Design.

6.3.3 Efficiency. The efficiency of the simulator runtime capability needs to be improved to allow it to efficiently simulate VLSI class designs. This can be accomplished by exploring the use of alternate algorithms (such as the use of a balanced tree algorithm for the event and transaction queues) and alternate data structures within the simulator kernel.

6.3.4 Runtime Control. The simulator runtime control should be enhanced to incorporate additional features such as a multiple breakpoint capability.

6.3.5 Report Capability. The output report capability should be expanded to provide the user with the capability to select optional output trace reports. The prototype simulator only allows output of an event trace report.

6.3.6 IEEE Standards. The fully implemented simulator should be designed to conform to IEEE VHDL standards and syntax.

6.4 Summary

VSIM was an important first step in the design and implementation of a complete VHDL simulator for the AFIT VHDL environment. VSIM provides proof of design concept for a VHDL simulator written in C and operating in a UNIX environment. It establishes a benchmark against which future development efforts can be evaluated.

Bibliography

1. d'Abreu, Michael. "Gate-Level Simulation," IEEE Design and Test, 2 (6): 63-71 (December 1985).

2. Moore, David H., and Wiliam, Toule J. "VHSIC's Industry Impace," 1981 University, Industry, Government Microelectronics Symposium. I-10-I-25. Mississippi, 1981.

3. Sumney, Larry W. "VHSIC, University, and Industry - Some Issues," 1981 University, Industry, Government Microelectronics Symposium, I-1 - I-9. Mississippi, 1981.

4. Shahad, Moe, et al. Computer, 18(2): 9-102 (February 1985).

5. Frauenfelder, Capt Deborah J. VHDL Language Analyzer. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1986.

6. Kamrowski, Capt Michael S. VHDL Hardware Simulator Using Parallel Processors. MS Thesis. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1986.

7. Chu, Yaohan, "Why Do We Need Hardware Description Languages?," IEEE, (December 1974).

8. Lipovski, J. H. Computer, 6(10), 14-17 (June 1977).

9. Booch, Grady. Software Engineering with Ada. Menlo Park, Ca. Benjamin Cummins Publishing Company, 1982.

10. Intermetrics, Inc. VHDL Design Analysis and Justification Report. Technical Report IR-MD-018-1, Bethesda, Md., 30 July 1984.

11. Intermetrics, Inc. Simulator Program Specification. U.S. Air Force Contract F33615-83-C-1003. Bethesda, Md., 30 July 1984.

12. Intermetrics, Inc. VHDL User's Manual: Volume 1-Tutorial. U.S. Air Force Contract F33615-83-C-1003. Bethesda, Md., 1 August 1985.

13. Intermetrics, Inc. VHDL Language Reference Manual: Version 7.2. U.S. Air Force Contract F33615-83-C-1003. Bethesda, Md., 1 August 1985.

14. Breuer, M. A. and A. Friedman. Diagnosis and Reliable Design of Digital Systems, Computer Science Press, Inc., Woodland Hills, Ca. 1976.

15. Kernighan, B. W. and Ritchie, D. M. The C Programming Language. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Appendix A:

Installation Guide

A.1 The UNIX Environment

The prototype simulator is designed to operate in the UNIX BSD 4.2 environment.

A.2 System

The simulator will execute on a DEC VAX.

A.3 Program Compilation

VSIM can be compiled and executed on the UNIX operating system using the system command 'make' and the VSIM makefile included in this appendix. Once the VSIM files are installed on a system, the 'make' command must be executed to create the executable file VSIM. VSIM can then be executed as explained in Appendix B without further use of the 'make' command. Use of the 'make' command is neccessary only after program modification.

```
FILES= vsim.c sim_initialize.c create_signal.c create_driver.c create_fanout.c \
        new_signal.c new_driver.c new_fanout.c debug_control.c create_eventq.c \
        simulate.c save.c debug_sig.c \
        debug_eventq.c debug_trans.c updateq.c post_trans.c get_func.c \
        behave1.c behave2.c behave3.c or_func.c not_func.c and_func.c\
        get_input_signals.c get_input_vectors.c pop_transact.c sig_addr.c\
        process_fanout.c debug_fanout.c post_fan_sig.c open_out_report.c\
        close_out_report.c convert.c  sim_structure.h

OBJECTS= vsim.o sim_initialize.o create_signal.o create_driver.o \
        create_fanout.o new_signal.o new_driver.o new_fanout.o debug_control.o\
        create_eventq.o simulate.o save.o debug_sig.o\
        debug_eventq.o debug_trans.o updateq.o post_trans.o get_func.o\
        behave1.o behave2.o behave3.o or_func.o not_func.o and_func.o\
        get_input_signals.o get_input_vectors.o pop_transact.o sig_addr.o\
        process_fanout.o debug_fanout.o post_fan_sig.o open_out_report.o\
        close_out_report.o convert.o

vsim: ${OBJECTS}
        ld -o vsim /lib/crt0.o ${OBJECTS} -lc -lg

vsim.o: vsim.c sim_structure.h
                cc -cg vsim.c

create_driver.o: create_driver.c sim_structure.h
                cc -cg create_driver.c

create_signal.o: create_signal.c sim_structure.h
                cc -cg create_signal.c

create_fanout.o: create_fanout.c sim_structure.h
                cc -cg create_fanout.c

new_driver.o: new_driver.c sim_structure.h
                cc -cg new_driver.c

new_signal.o: new_signal.c sim_structure.h
                cc -cg new_signal.c

new_fanout.o: new_fanout.c sim_structure.h
                cc -cg new_fanout.c

sim_initialize.o: sim_initialize.c sim_structure.h
                cc -cg sim_initialize.c

create_eventq.o: create_eventq.c sim_structure.h
                cc -cg create_eventq.c

debug_control.o: debug_control.c sim_structure.h
                cc -cg debug_control.c

simulate.o: simulate.c sim_structure.h
                cc -cg simulate.c
```

```
save.o: save.c
        cc -cg save.c

debug_sig.o: debug_sig.c sim_structure.h
        cc -cg debug_sig.c

debug_eventq.o: debug_eventq.c sim_structure.h
        cc -cg debug_eventq.c

debug_trans.o: debug_trans.c sim_structure.h
        cc -cg debug_trans.c

updateq.o: updateq.c sim_structure.h
        cc -cg updateq.c

post_trans.o: post_trans.c sim_structure.h
        cc -cg post_trans.c

get_func.o: get_func.c sim_structure.h
        cc -cg get_func.c

behave1.o: behave1.c sim_structure.h
        cc -cg behave1.c

behave2.o: behave2.c sim_structure.h
        cc -cg behave2.c

behave3.o: behave3.c sim_structure.h
        cc -cg behave3.c

or_func.o: or_func.c
        cc -cg or_func.c

not_func.o: not_func.c
        cc -cg not_func.c

and_func.o: and_func.c
        cc -cg and_func.c

get_input_signals.o: get_input_signals.c sim_structure.h
        cc -cg get_input_signals.c

get_input_vectors.o: get_input_vectors.c sim_structure.h
        cc -cg get_input_vectors.c

pop_transact.o: pop_transact.c sim_structure.h
        cc -cg pop_transact.c

sig_addr.o: sig_addr.c sim_structure.h
        cc -cg sig_addr.c

process_fanout.o: process_fanout.c sim_structure.h
        cc -cg process_fanout.c
```

```
debug_fanout.o: debug_fanout.c sim_structure.h
        cc -cg debug_fanout.c

post_fan_sig.o: post_fan_sig.c sim_structure.h
        cc -cg post_fan_sig.c

open_out_report.o: open_out_report.c sim_structure.h
        cc -cg open_out_report.c

close_out_report.o: close_out_report.c sim_structure.h
        cc -cg close_out_report.c

convert.o: convert.c sim_structure.h
        cc -cg convert.c
```

Appendix B:

Users Manual

B.1 Program Execution

VSIM allows the user the selection of several options through the command line. At a minimum, the command line must contain the program name VSIM and the name of the input vector file. The following format must be used for the command line:

VSIM [options] input file name

The command line options may be specified in any order and the program is insensitive to the number of allowable arguments which are present. The command line options are summarized below:

-d n Selects the debug option which causes the contents of selected structures to be printed to the output file. Debug prints the selected data the first time that simulate is entered and each time a transaction is processed. n can be one of four integer values (1, 2, 3, 4) where:

n = 1 prints all signal structures

n = 2 prints the event queue

n = 3 prints all transaction queues

n = 4 prints all data structures

-o filename Allows the user to select the output file where the output trace file will be written. The default is **sim output.**

-s n Allows the user to select the simulation start time. The time units specified must be the same as the units of time used for the input vector file and the circuit description. n must be an integer value. The default start time is 0.

-t n Allows the user to select the simulation termination time. The time units specified must be the same as the units of time used for the input vector file and the circuit description. n must be an integer value. The default termination time is 10000.

-b n Allows the user to select a breakpoint time when data will be dumped to the output file. Simulation continues after the breakpoint is processed. n must be an integer value. Not implemented on this prototype version.

B.2 Input Test Vector File

VSIM reads and interprets the input vector file. The user specifies the input filename on the command line. Failure to specify the input file will result in a fatal error and termination of program execution. VSIM accepts five valid input values: '0,' '1,' 'z' (high impedance),

'u' (unknown state) and '.' (means no change in previous value). A sample input vector file for VSIM is contained in Appendix C. The $ will be used as a delimiter between the designation of the input ports (signals) and the associated input vectors.

B.3 Timing

VSIM allows the user to select the simulation start or termination time as indicated above. Either one or both may be selected; however, a fatal error occurs if the termination time specified is less than the start time. Simulation time is not discrete but is event-directed, i.e., it is incremented according to the scheduled occurrence of events.

B.4 Initialization

Prior to the start of simulation, all the signal and driver values will be initialized to 'u,' the unknown state.

B.5 Error Checking

VSIM provides extensive error checking. If errors occur, they are displayed on either the standard output or written to the specified output file. An error on the command line or a failure to open the user specified input

vector file will result in a fatal error message displayed on the standard output. All other error messages are written to the specified output file. Errors can affect program execution in one of two ways: 1) after warnings program execution resumes, or 2) after fatal errors program execution is aborted and no output file is written.

B.6 Design Circuit Changes

Since the circuit design in VSIM is manually coded, the following modifications must be made to VSIM to simulate a different circuit design:

1) The input vector file must be modified to include new input signals and their values.

2) The three data files (signal, driver, and fanout) must be modified to include the new data for the circuits, signals, drivers and the signal's fanout.

3) The behavior functions need to be changed to accurately model the new circuit design. A behavior function is required for each logic gate in the circuit design. These changes can include:

   a) Changing the circuit delay type. Since inertial is the default in VHDL, transport delay must be set by declaring the global variable TRANSPORT to be TRUE.

   b) The circuit delay time can be set by the assignment of a value to the global variable **new time**.

   c) The input and output signals must be defined (# define) in the behavioral function.

   d) The generic function must be specified in the behavioral function. VSIM models the generic functions **and**, **or**, **not**. The user will have to create a new function to model other than one of these three gate functions.

Appendix C:

Sample Input Vector File

The following is a sample of an input vector file. A detailed explanation of the file can be found in section 3.5.4, Chapter 3.

```
alpha
b
c
$
10 1 . 0
13 u 0 1
15 1 z 0
17 . 0 1
20 0 1 .
30 1 u 0
40 0 . 1
45 1 1 0
65 0 . u
75 1 z 0
```

Appendix D:

Sample Signal Data File

The signal data file contains the signal name, mode and type for each signal in the circuit design.

```
alpha 1 2
b 1 2
c 1 2
delta 0  2
echo 2 2
foxtrot 2 2
```

Appendix E:

Sample Driver Data File

The driver data file contains the drivers for each signal in the circuit design. The first entry on each line is a number which tells the VSIM program how many drivers to read for a given signal.

```
1 alpha1
1 b1
1 c1
1 d1
1 e1
1 f1
```

Appendix F:

Sample Fanout Data File

The fanout data file contains the fanout (behavioral functions) for each signal in the circuit design. The first entry on each line is a number which tells the VSIM program the fanout for a given signal.

```
1 behave1
1 behave1
1 behave2
2 behave2 behave3
```

Appendix G:

Sample Output Trace File

VSIM produces an Output Event Trace File. The report provides the user with the following information:

1) Column 1 gives the time of the event.

2) Column 2 gives the signal which had the event.

3) Column 3 gives the value the signal had before the event.

4) Column 4 gives the new value of the signal.

5) Column 5 gives the source of the event: 1 indicating an input vector and 0 indicating a transaction created during the simulation.

6) A report summary providing simulation start and termination times, and total simulation runtime, transactions processed, events processed and transactions created.

AFIT VHDL Prototype Simulator Output Report

Events Processed

Time	Signal	Present Value	New Value	Transaction Type
1	alpha	u	0	1
1	b	u	0	1
1	c	u	0	1
15	delta	u	0	0
20	echo	u	1	0
25	foxtrot	u	0	0
26	c	0	1	1
26	b	0	1	1
40	delta	0	1	0
41	b	1	0	1
45	echo	1	0	0
50	foxtrot	0	1	0
55	delta	1	0	0
60	echo	0	1	0
65	foxtrot	1	0	0
66	b	0	1	1
80	delta	0	1	0
85	echo	1	0	0
90	foxtrot	0	1	0
91	c	1	0	1
91	b	1	0	1
101	foxtrot	1	0	0
105	delta	1	0	0
110	echo	0	1	0
116	alpha	0	1	1

Time	Signal	Present Value	New Value	Transaction Type
116	b	0	1	1
116	c	0	1	1
130	delta	0	1	0
135	echo	1	0	0
140	foxtrot	0	1	0

Simulation Summary

Simulation start time : 0

Simulation termination time :140

Total Simulation Run Time :140

Total Transactions Processed :40

Total Events Processed :30

Number of New Transactions Created :24

Appendix H:

VSIM Validation

This appendix contains the state table, boolean expressions, input vector file and output results used to validate the VSIM program.

STATE TABLE						
A	B	C	E	F	H	I
0	0	0	1	0	1	0
0	0	1	1	0	1	1
0	1	0	0	0	1	0
0	1	1	0	0	0	0
1	0	0	0	1	0	0
1	0	1	0	0	0	0
1	1	0	0	1	0	0
1	1	1	0	0	0	0

Expressions
$(A + B)' = E$
$(A + B)\ C = F$
$(A + BC)' = H$
$A'B'C = I$

```
alpha
b
c
$
1 0 0 0
26 0 1 1
41 0 0 1
66 0 1 1
91 0 0 0
116 1 1 1
```

AFIT VHDL Prototype Simulator Output Report

Events Processed

Time	Signal	Present Value	New Value	Transaction Type
1	alpha	u	0	1
1	b	u	0	1
1	c	u	0	1
15	delta	u	0	0
20	echo	u	1	0
25	foxtrot	u	0	0
26	c	0	1	1
26	b	0	1	1
40	delta	0	1	0
41	b	1	0	1
45	echo	1	0	0
50	foxtrot	0	1	0
55	delta	1	0	0
60	echo	0	1	0
65	foxtrot	1	0	0
66	b	0	1	1
80	delta	0	1	0
85	echo	1	0	0
90	foxtrot	0	1	0
91	c	1	0	1
91	b	1	0	1
101	foxtrot	1	0	0
105	delta	1	0	0
110	echo	0	1	0
116	alpha	0	1	1

Time	Signal	Present Value	New Value	Transaction Type
116	b	0	1	1
116	c	0	1	1
130	delta	0	1	0
135	echo	1	0	0
140	foxtrot	0	1	0

Simulation Summary

Simulation start time : 0

Simulation termination time :140

Total Simulation Run Time :140

Total Transactions Processed :40

Total Events Processed :30

Number of New Transactions Created :24

Appendix I:

Results

This appendix contains the results used for the analysis of VSIM presented in Chapter 5. These results were compiled from the summaries of VSIM Output Event Trace Files.

Results 10 Input Vectors

sum 10 3g

Simulation Summary

Simulation start time : 0

Simulation termination time :89

Total Simulation Run Time :89

Total Transactions Processed :50

Total Events Processed :32

Number of New Transactions Created :31

sum 10 6g

Simulation Summary

Simulation start time : 0

Simulation termination time :107

Total Simulation Run Time :107

Total Transactions Processed :75

Total Events Processed :41

Number of New Transactions Created :56

Results 20 Input Vectors

sum 20 3g

Simulation Summary

Simulation start time : 0

Simulation termination time :109

Total Simulation Run Time :109

Total Transactions Processed :65

Total Events Processed :41

Number of New Transactions Created :40

sum 20 6g

Simulation Summary

Simulation start time : 0

Simulation termination time :177

Total Simulation Run Time :177

Total Transactions Processed :140

Total Events Processed :76

Number of New Transactions Created :105

Results 30 Input Vectors

sum 30 g3

Simulation Summary

Simulation start time : 0

Simulation termination time :239

Total Simulation Run Time :239

Total Transactions Processed :139

Total Events Processed :89

Number of New Transactions Created :85

sum 30 6g

Simulation Summary

Simulation start time : 0

Simulation termination time :245

Total Simulation Run Time :245

Total Transactions Processed :198

Total Events Processed :108

Number of New Transactions Created :151

Results 60 Input Vectors

sum 60 g3

Simulation Summary

Simulation start time : 0

Simulation termination time :439

Total Simulation Run Time :439

Total Transactions Processed :280

Total Events Processed :182

Number of New Transactions Created :172

sum 60 6g

Simulation Summary

Simulation start time : 0

Simulation termination time :445

Total Simulation Run Time :445

Total Transactions Processed :395

Total Events Processed :219

Number of New Transactions Created :306

Results 120 Input Vectors

sum 120 g3

Simulation Summary

Simulation start time : 0

Simulation termination time :939

Total Simulation Run Time :939

Total Transactions Processed :559

Total Events Processed :366

Number of New Transactions Created :345

sum 120 6g

Simulation Summary

Simulation start time : 0

Simulation termination time :945

Total Simulation Run Time :945

Total Transactions Processed :786

Total Events Processed :441

Number of New Transactions Created :613

Results 240 Input Vectors

sum 240 g3

Simulation Summary

Simulation start time : 0

Simulation termination time :1939

Total Simulation Run Time :1939

Total Transactions Processed :1117

Total Events Processed :734

Number of New Transactions Created 691

sum 240 6g

Simulation Summary

Simulation start time : 0

Simulation termination time :1945

Total Simulation Run Time :1945

Total Transactions Processed :1569

Total Events Processed :885

Number of New Transactions Created :1227

Results 480 Input Vectors

sum 480 6g

Simulation Summary

Simulation start time : 0

Simulation termination time :3945

Total Simulation Run Time :3945

Total Transactions Processed :3135

Total Events Processed :1774

Number of New Transactions Created :2457

sum 480 g3

Simulation Summary

Simulation start time : 0

Simulation termination time :3939

Total Simulation Run Time :3939

Total Transactions Processed :2233

Total Events Processed :1470

Number of New Transactions Created :1384

Results 720 Input Vectors

sum 720 g3

Simulation Summary

Simulation start time : 0

Simulation termination time :5939

Total Simulation Run Time :5939

Total Transactions Processed :3348

Total Events Processed :2206

Number of New Transactions Created :2076

sum 720 6g

Simulation Summary

Simulation start time : 0

Simulation termination time :5945

Total Simulation Run Time :5945

Total Transactions Processed :4699

Total Events Processed :2662

Number of New Transactions Created :3685

Results 960 Input Vectors

sum 960 6g

Simulation Summary

Simulation start time : 0

Simulation termination time :7945

Total Simulation Run Time :7945

Total Transactions Processed :6265

Total Events Processed :3551

Number of New Transactions Created :4915

sum 960 g3

Simulation Summary

Simulation start time : 0

Simulation termination time :7939

Total Simulation Run Time :7939

Total Transactions Processed :4464

Total Events Processed :2942

Number of New Transactions Created :2769

Vita

William Leo Lynch, Jr. was born on May 18, 1947 in Somerville, Massachusetts. He graduated from Burlington High School in 1965 and received a Bachelor's degree from Northeastern University, Boston, Mass. in June 1969. Upon graduation, he was commissioned a second lieutenant in the U.S. Army through the ROTC program. He also received a Master's of Business Administration degree from the University of Utah in June 1982. His most recent assignment, from May 1985 to June 1986, was as the Executive Officer, 509th Signal Battalion, Camp Darby, Italy.

Permanent Address: 7 Olean Road
Burlington, Mass 01803

END

5-87

DTIC